

Développement sur Medulla

- [Comment étendre Medulla](#)
- [Comment utiliser l'API XML-RPC de Medulla](#)

Comment étendre Medulla

First of all, there are a few important things to note when developing around the agent:

The machine agent is automatically updated when the code is not identical between the agent and the agent codebase that is on the server. To disable this behaviour, the following setting needs to be added to agentconf.ini and the agent restarted:

```
[updateagent]
updating = 0
```

The plugins are also automatically updated if the version has changed. If a plugin needs to be modified, do not update the version until all the tests have been done.

There are four types of plugins for the machine agent: start plugins, update plugins, action plugins and scheduled plugins.

1. Start plugins are those run when the agent starts and are defined in start_machine.ini;
2. Update plugins are used to install or update external components used by the agent;
3. Action plugins are called by an action received by the agent;
4. Scheduled plugins are those called at a specific time or interval.

Each plugin can have its own config file named after the plugin name and needs to be added to the following parameter in agentconf.ini for the configuration to be loaded:

```
[plugin]
pluginlist = xxxxxxxx, yyyyyyy
```

The scheduled plugins have their own schedule defined in the plugin in the SCHEDULE parameter. This however can be overridden in the manage_scheduler_machine.ini file

There are 3 ways to extend Medulla:

1. interaction with the machine agent via a TCP socket
 2. interaction with the machine agent via named pipes
 3. machine agent action plugins
 4. machine agent scheduled plugins
-

How to interact with the agent via a TCP socket

A new action needs to be defined in **server_kiosk.py** in the function named **handle_client_connection** in the condition

```
if 'action' in result:
```

and this action added to the JSON message that is sent on the TCP socket.
Here is an example of the message sent:

```
{
  "action": "myNewAction",
  "sessionid": "mysessionid",
  "base64": false,
  "data": {
    "date": "2020-06-24T15:45:02.000Z",
    "family1": {
      "field1": "value1",
      "field2": "value2"
    },
    "family2": {
      "field1": "value1",
      "field2": "value2"
    }
  }
}
```

The above content is to be saved to a file named **json_file** to be sent via TCP socket or added to a variable named **json_message** in your code for sending via named pipes

And its counterpart in **handle_client_connection** function in **manage_kiosk_message** class.
In the following section:

```
try:
    _result = json.loads(minifyjsonstringrecv(msg))
```

Add

```
if _result['action'] == "myNewAction":
    substitute_recv = self.objectxmp.sub_monitoring
    logging.getLogger().warning("send to %s to %s" % (_result,substitute_recv ))
    self.objectxmp.send_message(mbody=json.dumps(_result),
                               mto=substitute_recv,
                               mtype='chat')

    return
```

And in the following loop:

```
if 'action' in result:
    if result['action'] == "kioskinterface":
    ...
```

Add

```
elif result['action'] == "myNewAction":
    datasend['action'] = "myNewSubstituteAction"
```

```
subs_recv = self.objectxmpp.sub_monitoring
datasend['sessionid'] = getRandomName(6, "mynews substituteaction")
datasend['data'] = result['data']
```

The above example will in turn send the message to sub_monitoring jid with a new action to be carried out: myNewSubstituteAction

The listening TCP port is defined in **agentconf.ini** parameter **kiosk/am_local_port**. Default value is 8765.

Here is an example TCP sender written in python for the next step which is the actual sending of the data to the TCP socket:

```
#!/usr/bin/env python
# -*- coding: utf-8; -*-
#
# (c) 2023 siveo, http://www.siveo.net
#
# This file is part of Medulla, http://www.siveo.net
#
# Pulse 2 is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# Pulse 2 is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with Medulla; if not, write to the Free Software
# Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
# MA 02110-1301, USA.
# file clientTCPcli.py

# Run python clientTCPcli.py -p ./file.json on the client machine to
# inject the data

from optparse import OptionParser

import socket
import sys
import os
import select

def send_message(message, host, port, timeout_in_seconds = 5):

    # Create a TCP/IP socket
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    # Connect the socket to the port where the server is listening
```

```

try:
    server_address = (host, port)
    print >>sys.stderr, 'connecting to %s port %s' % server_address
    sock.connect(server_address)
except socket.error , msgerror:
    print 'Bind failed on command interface ' + host + ' port ' + str(port) + ' Error Code :
    sys.exit(str(msgerror[0]))
try:
    # Send data
    print >>sys.stderr, 'sending "%s"' % message
    sock.sendall(message)

    # Look for the response
    amount_received = 0
    amount_expected = len(message)
    ready = select.select([sock], [], [], timeout_in_seconds)
    data = ""
    if ready[0]:
        data = sock.recv(4096)
        return 0, data
    return -1, ""
finally:
    print >>sys.stderr, 'closing socket'
    sock.close()

if __name__ == '__main__':
    optp = OptionParser()
    #optp.add_option("-h", "--help",action="store_true",
                    #dest="help", default=False,
                    #help="host")

    optp.add_option("-H", "--host",action="store_true",
                    dest="host", default="localhost",
                    help="host")

    optp.add_option("-P", "--port",
                    dest="port", default=8765,
                    help="port connection")

    optp.add_option("-T", "--timeout",
                    dest="timeout_in_seconds", default=2,
                    help="Recv timeout in seconds")

    optp.add_option("-m", "--msg",
                    dest="msg", default = "",
                    help="message for sending to TCP server")

    optp.add_option("-p", "--pathfile",
                    dest="pathfile", default = None,
                    help="content File for sending to TCP server")

    opts, args = optp.parse_args()

```

```

#if opts.help:
    #print "usage commande"
    #os._exit(0)
message = ""
if opts.pathfile is None:
    if opts.msg == "":
        print "option missing < -m | -p> "
        sys.exit(-1)
    message = opts.msg
else:
    if os.path.exists(opts.pathfile):
        with open(opts.pathfile, 'r') as f:
            message = f.read()

if message != "":
    code, msg = send_message(message, opts.host, opts.port, opts.timeout_in_seconds)
print "code error %s, reponse server %s"%(code,msg)

```

To send the message:

```
python clientTCPcli.py -p <json_file>
```

How to interact with the agent via named pipes

The way to interact with the agent via named pipes is done the same way as for interacting with the agent via TCP socket except for sending the message.

To send the message to the named pipe here is an example code written in Python:

```

import win32file

def send_message(json_message):
    fileHandle = win32file.CreateFile("\\\\.\\pipe\\interfacechang",
                                      win32file.GENERIC_READ | win32file.GENERIC_WRITE,
                                      0,
                                      None,
                                      win32file.OPEN_EXISTING,
                                      0,
                                      None)

    win32file.WriteFile(fileHandle, json_message)
    win32file.CloseHandle(fileHandle)

```

How to write action plugins for the agent

Below is a template that can be used for writing action plugins:

```
import logging
import json

plugin = {"VERSION": "1.0", "NAME": "mynewaction", "TYPE": "machine"}

logger = logging.getLogger()

def action( objectxmpp, action, sessionid, data, message, dataerreur):
    logger.debug("#####")
    logger.debug("call %s from %s session id %s" % (plugin, message['from'], sessionid))
    logger.debug("#####")
    datasend = {"action" : "myNewSubstituteAction",
                "data" : data,
                "sessionid": sessionid,
                "ret": 0,
                "base64": False
               }
    objectxmpp.send_message(mto=objectxmpp.sub_monitoring,
                           mbody=json.dumps(datasend),
                           mtype='chat')
```

Please note the following:

- NAME must match the name of the plugin file. Here the file will be named `plugin_mynewaction.py`
- TYPE must be defined to machine, relayservice or all depending on its target
- the action function will be the code executed by default. The above example will in turn send the message to sub_monitoring jid with a new action to be carried out: `myNewSubstituteAction`

How to write scheduled plugins for the agent

Below is a template that can be used for writing scheduled plugins:

```
import logging
import json
import os
import ConfigParser
from pulse_xmpp_agent.lib.agentconffile import directoryconffile
from pulse_xmpp_agent.lib.utils import file_put_contents

plugin = {"VERSION": "1.0", "NAME": "scheduling_mynewscheduledaction", "TYPE": "machine", "SCHEDULE": "*/15 * * * *"}

SCHEDULE = {"schedule" : "*/15 * * * *", "nb" : -1}

logger = logging.getLogger()
```

```

def schedule_main(xmppobject):
    logger.debug("=====")
    logger.debug(plugin)
    logger.debug("=====")
    if xmppobject.num_call_scheduling_mynewscheduledaction == 0:
        __read_conf(xmppobject)

    if xmppobject.config.mynewscheduledaction_enable:
        data = {}
        data['family1'] = {}
        data['family1']['field1'] = "value1"
        data['family1']['field2'] = "value2"
        data['family2'] = {}
        data['family2']['field1'] = "value1"
        data['family2']['field2'] = "value2"

    if xmppobject.config.mynewscheduledaction_forward:
        datasend = {"action" : "myNewSubstituteAction",
                    "data" : data,
                    "sessionid": "mysessionid",
                    "base64": False
                   }
        objectxmpp.send_message(mto=objectxmpp.sub_monitoring,
                               mbody=json.dumps(datasend),
                               mtype='chat')

def __read_conf(xmppobject):
    """
    Read the plugin configuration
    """
    configfilename = os.path.join(directoryconffile(), "%s.ini" % plugin['NAME'])
    logger.debug("Reading configuration in file %s" % configfilename)

    #default parameters
    xmppobject.config.mynewscheduledaction_enable = True
    xmppobject.config.mynewscheduledaction_forward = False

    if not os.path.isfile(configfilename):
        logger.warning("Plugin %s configuration file %s missing" % (plugin['NAME'], configfilename))
        logger.warning("The missing configuration file will be created automatically.")
        file_put_contents(configfilename,
                          "[mynewscheduledaction]\n" \
                          "enable = 1\n" \
                          "forward = 0\n")

    # Load configuration from file
    Config = ConfigParser.ConfigParser()
    Config.read(configfilename)
    if os.path.exists(configfilename + ".local"):
        Config.read(configfilename + ".local")
    if Config.has_section("mynewscheduledaction"):
        if Config.has_option("mynewscheduledaction", "enable"):
            xmppobject.config.mynewscheduledaction_enable = Config.getboolean('mynewscheduledact

```

```
if Config.has_option("mynewscheduledaction", "forward"):
    xmppobject.config.mynewscheduledaction_forward = Config.getboolean('mynewscheduledac
```

Please note the following:

- NAME must match the name of the plugin file. Here the file will be named plugin_mynewaction.py
- TYPE must be defined to machine, relayserver or all depending on its target
- SCHEDULE notation is similar to the cron notation. The extra parameter nb defines how many times the plugin must run. If -1, it will run forever
- the schedule_main function will be the code executed by default. The above example will read a config file or create it if it does not exist and send a message to sub_monitoring jid with a new action to be carried out: myNewSubstituteAction

Comment utiliser l'API XML-RPC de Medulla

This will guide the user to using Medulla's XML-RPC API : [Medulla's XML-RPC API.php](#)

1: Configuration

Check the following:

- SSL Certificate: Grab the certificate from Medulla server:
http://<medulla_server>/downloads/medulla-ca-chain.cert.pem
- Install the certificate on the server that will be querying the API
 - For Debian based systems, copy the file to `/usr/local/share/ca-certificates` and change its extension to `crt`, then import the certificate by running `update-ca-certificates` command
 - For RedHat based systems, copy the file to `/etc/pki/ca-trust/source/anchors` and change its extension to `crt`, then import the certificate by running `update-ca-trust extract` command
- If needed, add the ip and hostname of the XML-RPC server to `/etc/hosts`

2: Using the API

First authenticate the user and fetch the session cookie:

```
/**
 * Exécute une requête XML-RPC.
 *
 * @param string $method Le nom de la méthode XML-RPC à appeler.
 * @param array $params Les paramètres à passer à la méthode.
 * @param bool $includeCookie Indique si le cookie de session doit être inclus dans la requête.
 * @return array Un tableau contenant les en-têtes HTTP et le corps de la réponse.
 * @throws Exception Si une erreur se produit lors de la connexion ou de l'envoi de la requête.
 */
function executeRequest($method, $params, $includeCookie = false) {
    $agentInfo = $_SESSION["XMLRPC_agent"];
    $requestXml = xmlrpc_encode_request($method, $params, ['output_type' => 'php', 'verbosity' => 0]);

    // On définit les en-têtes HTTP
    $url = "/";
    $httpQuery = "POST " . $url . " HTTP/1.0\r\n";
    $httpQuery .= "User-Agent: MMC web interface\r\n";
    $httpQuery .= "Host: " . $agentInfo["host"] . ":" . $agentInfo["port"] . "\r\n";
    $httpQuery .= "Content-Type: text/xml\r\n";
```

```

$httpQuery .= "Content-Length: " . strlen($requestXml) . "\r\n";
// On ajoute le cookie si nécessaire
if ($includeCookie) {
    $httpQuery .= "Cookie: " . $_SESSION['cookie'] . "\r\n";
}
$httpQuery .= "Authorization: Basic " . base64_encode($agentInfo["login"] . ":" . $agentInfo["password"]);
$httpQuery .= $requestXml;

// Configurer le contexte SSL
// 'allow_self_signed' est défini sur false pour n'accepter que les certificats signés par u
// 'verify_peer' est défini sur true pour vérifier le certificat SSL du serveur
$context = stream_context_create();
$proto = $agentInfo["scheme"] == "https" ? "ssl://" : "";
if ($proto) {
    stream_context_set_option($context, "ssl", "allow_self_signed", false);
    stream_context_set_option($context, "ssl", "verify_peer", true);
}

// On ouvre la connexion au serveur
$socket = stream_socket_client($proto . $agentInfo["host"] . ":" . $agentInfo["port"], $errno, $errstr);
if (!$socket) {
    throw new Exception("Unable to connect to XML-RPC server: $errstr ($errno)");
}

// On ajoute un délai de 60 secondes pour la lecture
stream_set_timeout($socket, 60);

if (!fwrite($socket, $httpQuery)) {
    throw new Exception("Unable to send data to XML-RPC server");
}

$responseXml = '';
while (!feof($socket)) {
    $ret = fgets($socket, 128);
    $responseXml .= $ret;
}
fclose($socket);

// Séparez les en-têtes HTTP du corps de la réponse
list($headers, $body) = explode("\r\n\r\n", $responseXml, 2);
return [$headers, $body];
}

/**
 * Authentifie l'utilisateur et récupère le cookie de session.
 * @param string $method Le nom de la méthode XML-RPC à appeler.
 * @param array $params Les paramètres à passer à la méthode.
 * @return string Le cookie de session.
 * @throws Exception Si une erreur se produit lors de l'authentification.
 */
function authenticateAndGetCookie($method, $params) {
    list($headers, $body) = executeRequest($method, $params);
}

```

```

// Parsing des en-têtes HTTP pour extraire le cookie de session
$headers_array = array();
$header_lines = explode("\r\n", $headers);
foreach ($header_lines as $header) {
    $parts = explode(':', $header, 2);
    if (count($parts) == 2) {
        $headers_array[$parts[0]] = $parts[1];
    }
}

// Utilisation des en-têtes analysés
if (isset($headers_array['Set-Cookie'])) {
    $cookie = $headers_array['Set-Cookie'];
    $_SESSION['cookie'] = $cookie;
} else {
    throw new Exception('Authentication failed, no cookie received');
}

return $cookie;
}

```

3: Examples of requests

- Get the list of all packages

```

/**
 * Envoie une requête XML-RPC et retourne la réponse uniquement si authentifié par cookie.
 * @param string $method Le nom de la méthode XML-RPC à appeler.
 * @param array $params Les paramètres à passer à la méthode.
 * @return array La réponse XML-RPC.
 * @throws Exception Si une erreur se produit lors de l'envoi de la requête.
 */
function sendXmlRpcRequest($method, $params) {
    list($headers, $body) = executeRequest($method, $params, true);
    $responseXml = substr($body, strpos($body, '<?xml'));
    $response = xmlrpc_decode($responseXml, 'UTF-8');
    if (is_array($response) && xmlrpc_is_fault($response)) {
        throw new Exception
("XML-RPC fault: {$response['faultString']} ({$response['faultCode']}");
    }
    return $response;
}

$method = "pkgs.get_all_packages";
$params = [
    'root', // login
    false, // sharing_activated
    0,     // start
    10,    // end
    [
        'filter' => 'hostname', // example name of package
    ]
]

```

```

    ] // ctx
];

try {
    $responseXml = sendXmlRpcRequest($method, $params);

} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
}

```

Tips: To get the package UUID: \$responseXml['datas']['uuid'][\$key]

- Get the list of all machines

```

$method = "xmppmaster.get_machines_list";
$params = [
    0, // start
    20, // end
    [
        'filter' => '',
        'field' => 'allchamp',
        'computerpresence' => 'presence', // Définir si on veut les machines présentes
        // 'computerpresence' => 'no_presence', // Définir si on veut les machines non présentes
        'location' => "UUID0", // glpi_id - entity
    ] // ctx
];

try {
    $responseXml = sendXmlRpcRequest($method, $params);
} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
}

```

- Get a machine's details

```

$method = "glpi.getLastMachineInventoryPart";
$params = [
    $uuid,
    'Summary',
    0, // minbound
    0, // maxbound
    "", // filter
    [
        "hide_win_updates" => false,
        "history_delta" => false
    ], // options
];

try {
    $responseXml = sendXmlRpcRequest($method, $params);
}

```

```
} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
}
```

Tips: To get the UUID of the machine: `$reponseXml['data']['uuid_inventorymachine'][$key]`. This also corresponds to GLPI Machine ID prefixed by UUID

- Deploy a specific package on a target machine

```
$method = "msc.add_command_api";
$pid = "96982fce-hostname_vq918j7wtwzm610by"; // pid - paquet_id
$target = "UUID1"; // target - uuid_machine

$params = [
    $pid, // pid - paquet_id
    $target, // target - uuid_machine
    array( // params
        "name" => "devdemo-win-1",
        "hostname" => "devdemo-win-1",
        "uuid" => $target,
        "gid" => NULL,
        "from" => "base|computers|msctabs|tablogs",
        "pid" => $pid,
        // "ltitle" => "TITRE DU DEPLOIEMENT",
        "create_directory" => "on",
        "start_script" => "on",
        "clean_on_success" => "on",
        "do_reboot" => "",
        "do_wol" => "",
        "do_inventory" => "on",
        "next_connection_delay" => "60",
        "max_connection_attempt" => "3",
        "maxbw" => "0",
        "deployment_intervals" => "",
        "tab" => "tablaunch",
        "issue_halt_to" => array(),
    ),
    "push", // mode
    NULL, // gid
    array(), // proxy
    0 // cmd_type
];

try {
    $responseXml = sendXmlRpcRequest($method, $params);

    $commandId = $responseXml; // Récupérer l'ID de la commande

    $method2 = "xmppmaster.addlogincommand";
    $params2 = [
        'root', // login
        $commandId, // commandid
    ];
}
```

```

        '', // grpId
        '', // nb_machine_in_grp
        '', // instructions_nb_machine_for_exec
        '', // instructions_datetime_for_exec
        '', // parameterspackage
        0, // rebootrequired
        0, // shutdownrequired
        0, // bandwidth
        0, // syncting
        [] // params
    ];

    try {
        $responseXml2 = sendXmlRpcRequest($method2, $params2);

    } catch (Exception $e) {
        echo "Error: " . $e->getMessage();
    }

} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
}
}

```

- Get deployment logs (Audit) from sessionname

```

$method = "xmpmaster.getlineLogsession";
$params = [
    "command63bb5ee8fc834eae89" // sessionname
];

try {
    $responseXml = sendXmlRpcRequest($method, $params);
} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
}

```

Tips: To get the deployment sessionname: `$responseXml['tabdeploy']['sessionid'][$key]`

- Get all deployment logs by User and/or target

```

$method = "xmpmaster.get_deploy_by_user_with_interval";
$params = [
    "root", // login_user
    "", // state_deploy
    86400, // intervalsearch
    0, // start_pagination
    "20", // end_pagination
    "spo-win-1", // filt - hostname_machine
    "command" // typedeploy
];

```

```
try {  
    $responseXml = sendXmlRpcRequest($method, $params);  
} catch (Exception $e) {  
    echo "Error: " . $e->getMessage();  
}
```

[New boost](#)