How to extend 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 on the agent does not match the agent codebase on the server. To disable this behavior, the following setting must 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 tests have been completed.

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 triggered 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 configuration file named after the plugin name and must be added to the following parameter in agentconf.ini for the configuration to be loaded:

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

The scheduled plugins have their own schedule defined in the plugin under the SCHEDULE parameter. However, this can be overridden in the manage_scheduler_machine.ini file

---

There are three 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 must be defined in**server_kiosk.py**within the function named **handle_client_connection**under the condition

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

and this action added to the JSON message sent over 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 the**handle_client_connection**function in the**manage_kiosk_message** class.
In the following section:

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

Add

```
if _result['action'] == "myNewAction":
    substitute_recv = self.objectxmpp.sub_monitoring
    logging.getLogger().warning("send to %s to %s" % (_result, substitute_recv))
    self.objectxmpp.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, "mynewsubstituteaction")
    datasend['data'] = result['data']
```

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

The listening TCP port is defined in**the agentconf.ini parameterkiosk/am_local_port**. The 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="connection port")

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

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

```
    optp.add_option("-p", "--pathfile",
                        dest="pathfile", default = None,
                            help="content file to send to TCP server")

    opts, args = optp.parse_args()
    #if opts.help:
        #print "command usage"
        #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 "error code %s, server response %s"%(code,msg)
```

To send the message:

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

How to interact with the agent via named pipes

Interacting with the agent via named pipes is done the same way as interacting with the agent via a 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 set to machine, relayserver, or all, depending on its target
- The action function will be the code executed by default. The example above will send the message to the sub_monitoring JID with a new action to be performed: 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", "SCHED

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'], configfilena
        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('mynewscheduleda
```

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 set to machine, relayserver, or all, depending on its target
- SCHEDULE notation is similar to cron notation. The additional parameter nb defines how many times the plugin must run. If set to -1, it will run indefinitely
- The schedule_main function will be the code executed by default. The example above will read a configuration file or create it if it does not exist, and send a message to the sub_monitoring JID with a new action to be performed: myNewSubstituteAction

---

Revision #2
Created 2026-04-29 19:03:47 UTC by Adrien Thaissen
Updated 2026-04-29 19:24:23 UTC by Adrien Thaissen