

MuPIF.org Platform User Manual

Authors: B. Patzák¹ and V. Šmilauer¹

Version 2.3 - 02/2020

¹ Czech Technical University, Faculty of Civil Engineering, Department of Mechanics, Thákurova 7, 16629, Prague, Czech Republic.

1. Table of Content

[1. Table of Content](#)

[2. Introduction](#)

[3. Platform installation](#)

[3.1. Prerequisites](#)

[3.1.1. Windows platforms](#)

[3.1.2. Linux / Unix \(*nix\) platforms](#)

[3.1.3. General requirements](#)

[3.1.4. Other recommended packages/software](#)

[3.2. Installing the MuPIF platform](#)

[3.3. Verifying platform installation](#)

[3.3.1. Running unit tests](#)

[3.3.2. Running examples](#)

[4. Platform operations](#)

[5. Platform APIs](#)

[5.1. Common API for all components](#)

[5.1.1 Metadata and metadata schemas](#)

[5.2. Model class](#)

[5.3. Workflow class](#)

[5.4. Property class](#)

[5.5. Field class](#)

[5.6. Function class](#)

[5.7. TimeStep class](#)

[5.8. Mesh class](#)

[5.9. Cell class](#)

[5.10. Vertex class](#)

[5.11. BoundingBox](#)

[5.12. APIError](#)

[6. Developing Application Program Interface \(API\)](#)

[7. Developing user workflows](#)

[7.1. Workflow templates](#)

[7.2. Workflow example](#)

[7.3. Workflow as a class](#)

[8. Distributed Model](#)

- [8.1. Distributed aspects of the API](#)
- [8.2. Requirements for distributed computing](#)
- [8.3. Internal platform solution - JobManager resource allocation](#)
 - [8.3.1. Setting up a Job Manager](#)
 - [8.3.2. Configuration](#)
- [8.4. Securing the communication using SSH tunnels](#)
- [8.5. Setting up ssh server](#)
- [8.6. Example of distributed scenario with ssh tunneling](#)
- [8.7. Advanced SSH setting](#)
- [8.8. Using Virtual Private Network \(VPN\)](#)
 - [8.8.1. Generalities](#)
 - [8.8.2. Setup](#)
 - [8.8.3. Example of simulation scenario using VPN](#)
 - [8.8.4. Online Monitoring tool](#)

[9. Acknowledgements](#)

[10. ChangeLog](#)

- [10.1. Version 1.1 \(05/2017\)](#)
- [10.2. Version 2.0.0 \(12/2017\)](#)
- [10.3. Version 2.2.0 \(04/2019\)](#)
- [10.4. Version 2.3.0 \(02/2020\)](#)

[11. References](#)

2. Introduction

MuPIF (www.mupif.org) is an integration framework, that facilitates the implementation of multi-physic and multi-level simulation workflows, built from independently developed components. MuPIF is open source, distributed under LGPL license.

The approach followed in the MuPIF is based on an system of distributed, interacting objects designed to solve given problem. The individual objects represent entities in the problem domain, including individual simulation packages, but also the data, such as fields and properties. The abstract classes are introduced for all entities in the model space [1]. They define a common interface, called API, that needs to be implemented by any derived class, representing particular implementation of specific component. Such interface concept allows using any derived class on a very abstract level, using common services defined by abstract class, without being concerned with the implementation details of an individual software component. The APIs have been developed not only for individual models, but also for simulation data, like spatial fields, properties, etc.

The complex simulation pipeline developed in MuPIF-platform consists of top-level script in Python language [3] (called scenario) enriched by newly introduced classes. Later in the project, the top level script will be generated using a graphical tool. In principle, any control script can be recast into a class implementing Model class interface, so that it could itself represent an application in MuPIF platform. Such an approach would allow building a hierarchy of nested applications. The application steering and data exchange will be realized in a standard way by calling individual services (methods). In case of distributed environments, a transparent communication layer is provided, as described in the subsection on Distributed environments. The software design of the platform has been described in [5,6,7].

Even though the platform can be used locally on a single computer orchestrating installed applications, the real strength of the MuPIF platform is its distributed design, allowing to execute simulation scenarios involving remote applications. The concept of so called proxy object that represent remote objects allows to hide all the details of remote data exchange and execution to the user. In turn, only minimal change of local simulation scenarios is required when distributed resources are included. The distributed model is described in Section [Distributed Model](#).

3. Platform installation

3.1. Prerequisites

3.1.1. Windows platforms

- We suggest to install Anaconda scientific python package, which includes Python ≥ 3.4 , <https://store.continuum.io/cshop/anaconda/>

- For secure communication either VPN client/server or SSH client/server is needed (recommended client: putty.exe [<http://www.putty.org/>], ssh server: FreeSSHd [<http://www.freesshd.com/>]).

3.1.2. Linux / Unix (*nix) platforms

- The Python (Python ≥ 3.4) installation is required.
- You can download the python installation package from <https://www.python.org/downloads/>. Just pick up the latest version in the 3.x series (tested version 3.5.2).
- We recommend to install *pip3* - a tool for installing and managing Python3 packages. If not already installed as a part of your python distribution, the installation instructions can be found [here](#).
- For secure communication either VPN client/server (openVPN recommended [<https://openvpn.net/index.php/open-source/downloads.html>]) or SSH client/server is needed (usually included in standard distribution).

3.1.3. General requirements

- MuPIF platform depends/requires, besides others, Pyro4 and numpy modules. They can be installed separately for a particular system or using *pip3*. If you install the whole MuPIF package, it takes care automatically for all dependencies. However, using *git* repository requires those Python modules to be installed separately. For example, to install Pyro4 version 4.75:

```
pip3 install Pyro4==4.75
```

- MuPIF platform requires pyvtk (tested 0.4.85) python module. To install this module using *pip*:

```
pip3 install pyvtk
```

- MuPIF requires enum34 module, which can be installed also using *pip*:

```
pip3 install enum34
```

3.1.4. Other recommended packages/software

- Paraview (tested 4.2.0), visualization application for vtu data files, <http://www.paraview.org/>
- Windows: Notepad++ (tested 6.6.9), <http://notepad-plus-plus.org/>
- Windows: conEmu, windows terminal emulator, <https://code.google.com/p/conemu-maximus5/>

3.2. Installing the MuPIF platform

The recommended procedure is to install platform as a python module using *pip3*:

```
pip3 install mupif
```

This type of installation automatically satisfies all the dependencies.

Alternatively, the development version of the platform can be installed from *git* repository:

- We recommend to install git, a open source revision control tool. You can install git using your package management tool or download installation package directly from [git website](#).
- Once you have git installed, just clone the MuPIF platform repository into a directory "mupif-code":

```
git clone https://github.com/mupif/mupif.git mupif
```

3.3. Verifying platform installation

3.3.1. Running unit tests

MuPIF platform comes with unit tests. To run unit tests we recommend to install *nose* python module, which facilitates automatic discovery and execution of individual tests. To install nose module using pip:

```
pip3 install nose
```

This will install the nose libraries, as well as the [nosetests](#) script, which can be used to execute the unit tests. From top level MuPIF installation directory enter:

```
cd tests
nosetests -v
```

You should see output something like this:

```
test_containsPoint (mupif.tests.test_BBox.BBox_TestCase) ... ok
test_intersects (mupif.tests.test_BBox.BBox_TestCase) ... ok
test_merge (mupif.tests.test_BBox.BBox_TestCase) ... ok
test_containsPoint (mupif.tests.test_Cell.Triangle_2d_lin_TestCase) ... ok
test_geometryType (mupif.tests.test_Cell.Triangle_2d_lin_TestCase) ... ok
test_glob2loc (mupif.tests.test_Cell.Triangle_2d_lin_TestCase) ... ok
test_interpolate (mupif.tests.test_Cell.Triangle_2d_lin_TestCase) ... ok
.....
testOctreeNotPickled (mupif.tests.test_saveload.TestSaveLoad) ... ok
-----
Ran 82 tests in 2.166s

OK
```

Indicating that *nose* found and ran listed tests successfully.

3.3.2. Running examples

In addition, the platform installation comes with many examples, that can be used to verify the successful installation as well, but they also serve as an educational examples illustrating how to use the platform. The examples are located in examples subfolder. For example, to run Example01:

```
cd examples/Example01
python3 Example01.py
```

4. Platform operations

The complex simulation pipeline developed in MuPIF-platform consists of top-level script in Python language (called scenario) enriched by newly introduced classes. These classes represent fundamental entities in the model space (such as simulation tools, properties, fields, solution steps, interpolation cells, units, etc). The top level classes are defined for these entities, defining a common interface allowing to manipulate individual representations using a single common interface. The top level classes and their interface is described in platform Interface Specification document [1].

In this document, we present a simple, minimum working example, illustrating the basic concept. The example presented in this section is assumed to be executed locally. How to extend these examples into distributed version is discussed in the section [8. Distributed Model](#).

The presented example in Listing 1 illustrates an example of so called weak-coupling, where for each solution step, the first application (Application1) evaluates the value of concentration that is passed to the second application (Application2) which, based on provided concentration values (PropertyID.PID_Concentration), evaluates the average cumulative concentration (PropertyID.PID_CumulativeConcentration). This is repeated for each solution step. The example also illustrates, how solution steps can be generated in order to satisfy time step stability requirements of individual applications.

```

from mupif import *
import application1
import application2

time = 0
timestepnumber=0
targetTime = 1.0

app1 = application1.application1(None) # create an instance of application #1
app2 = application2.application2(None) # create an instance of application #2
app1.initialize() #initialize app1
app2.initialize() #initialize app2

# loop over time steps
while (abs(time -targetTime) > 1.e-6):
    #determine critical time step
    dt2 = app2.getCriticalTimeStep()
    dt = min(app1.getCriticalTimeStep(), dt2)
    #update time
    time = time+dt
    if (time > targetTime):
        #make sure we reach targetTime at the end
        time = targetTime
    timestepnumber = timestepnumber+1

    # create a time step
    istep = TimeStep.TimeStep(time, dt, timestepnumber)

    try:
        #solve problem 1
        app1.solveStep(istep)
        #request temperature field from app1
        c = app1.getProperty(PropertyID.PID_Concentration, istep)
        # register temperature field in app2
        app2.setProperty (c)
        # solve second sub-problem
        app2.solveStep(istep)
        prop = app2.getProperty(PropertyID.PID_CumulativeConcentration, istep)
        print ("Time: %5.2f concentraion %5.2f, running average %5.2f" %
              (istep.getTime(), c.getValue(), prop.getValue()))

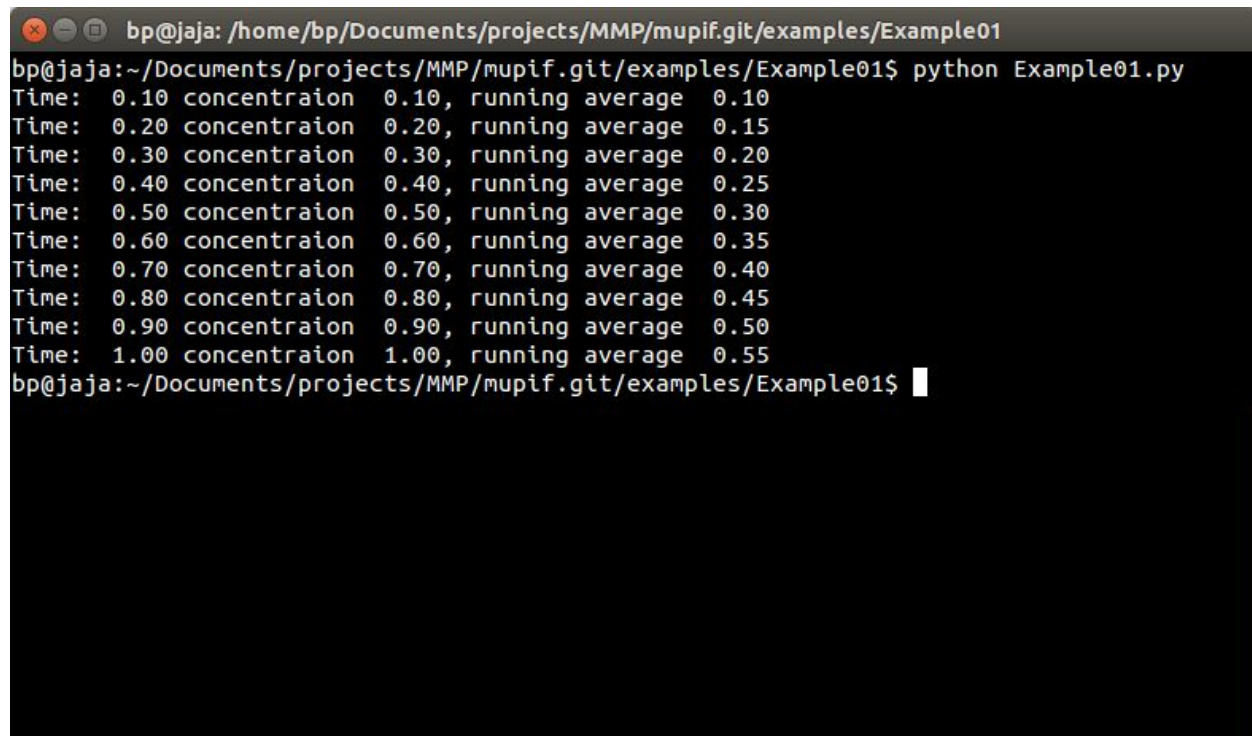
    except APIError.APIError as e:
        logger.error("Following API error occurred: %s" % e )
        break

# terminate
app1.terminate();
app2.terminate();

```

Listing 1: Simple example illustrating simulation scenario

The full listing of this example can be found in [examples/Example01](#). The output is illustrated in Fig. 1.

A terminal window with a dark background and light-colored text. The window title is 'bp@jaja: /home/bp/Documents/projects/MMP/mupif.git/examples/Example01'. The prompt is 'bp@jaja:~/Documents/projects/MMP/mupif.git/examples/Example01\$'. The command 'python Example01.py' has been executed, resulting in ten lines of output. Each line shows a time value from 0.10 to 1.00 in increments of 0.10, followed by the word 'concentraion' (misspelled), the same time value, a comma, the words 'running average', and a value that increases from 0.10 to 0.55 in increments of 0.05. The prompt 'bp@jaja:~/Documents/projects/MMP/mupif.git/examples/Example01\$' is shown again at the bottom with a cursor.

```
bp@jaja: /home/bp/Documents/projects/MMP/mupif.git/examples/Example01
bp@jaja:~/Documents/projects/MMP/mupif.git/examples/Example01$ python Example01.py
Time:  0.10 concentraion  0.10, running average  0.10
Time:  0.20 concentraion  0.20, running average  0.15
Time:  0.30 concentraion  0.30, running average  0.20
Time:  0.40 concentraion  0.40, running average  0.25
Time:  0.50 concentraion  0.50, running average  0.30
Time:  0.60 concentraion  0.60, running average  0.35
Time:  0.70 concentraion  0.70, running average  0.40
Time:  0.80 concentraion  0.80, running average  0.45
Time:  0.90 concentraion  0.90, running average  0.50
Time:  1.00 concentraion  1.00, running average  0.55
bp@jaja:~/Documents/projects/MMP/mupif.git/examples/Example01$
```

Figure 1: Output from Example01.py

The platform installation comes with many examples, located in *examples* subdirectory of platform installation and also accessible [online](#) in the platform repository. They illustrate various aspects, including field mapping, vtk output, etc.

5. Platform APIs

In this chapter are presented the abstract interfaces (APIs) of abstract classes that have been designed to represent basic building blocks of the complex multi-physics simulations, including individual simulation packages, but also the high level complex data (such as spatial fields and properties). The abstract base classes are defined for all relevant entities, see Fig. 2. Their primary role is to define abstract interfaces (APIs), which allow manipulating individual objects using generic interface without being concerned by internal details of individual instances. One of the key and distinct features of the MuPIF platform is that such an abstraction (defined by top level classes) is not only developed for individual models, but also defined for the simulation data themselves. The focus is on services provided by objects and not on underlying data. The object representation of data encapsulates the data themselves, related metadata, and related algorithms. Individual models then do not have to interpret the complex data themselves; they receive data and algorithms in one consistent package. This also allows the platform to be independent of particular data format, without requiring any changes on the model side to work with new format.

In the rest of this section, the individual abstract classes and their interfaces are described in detail. For each class a table is provided, where on the left column the individual services and their arguments are presented, following the Pydoc [7] syntax. In the right column, the description of individual service is given, input arguments are described (denoted by ARGS) including their type (in parenthesis). The return values are described in a similar way (denoted by Returns). More extensive documentation of MuPIF abstract classes exists in MuPIF documentation [8].

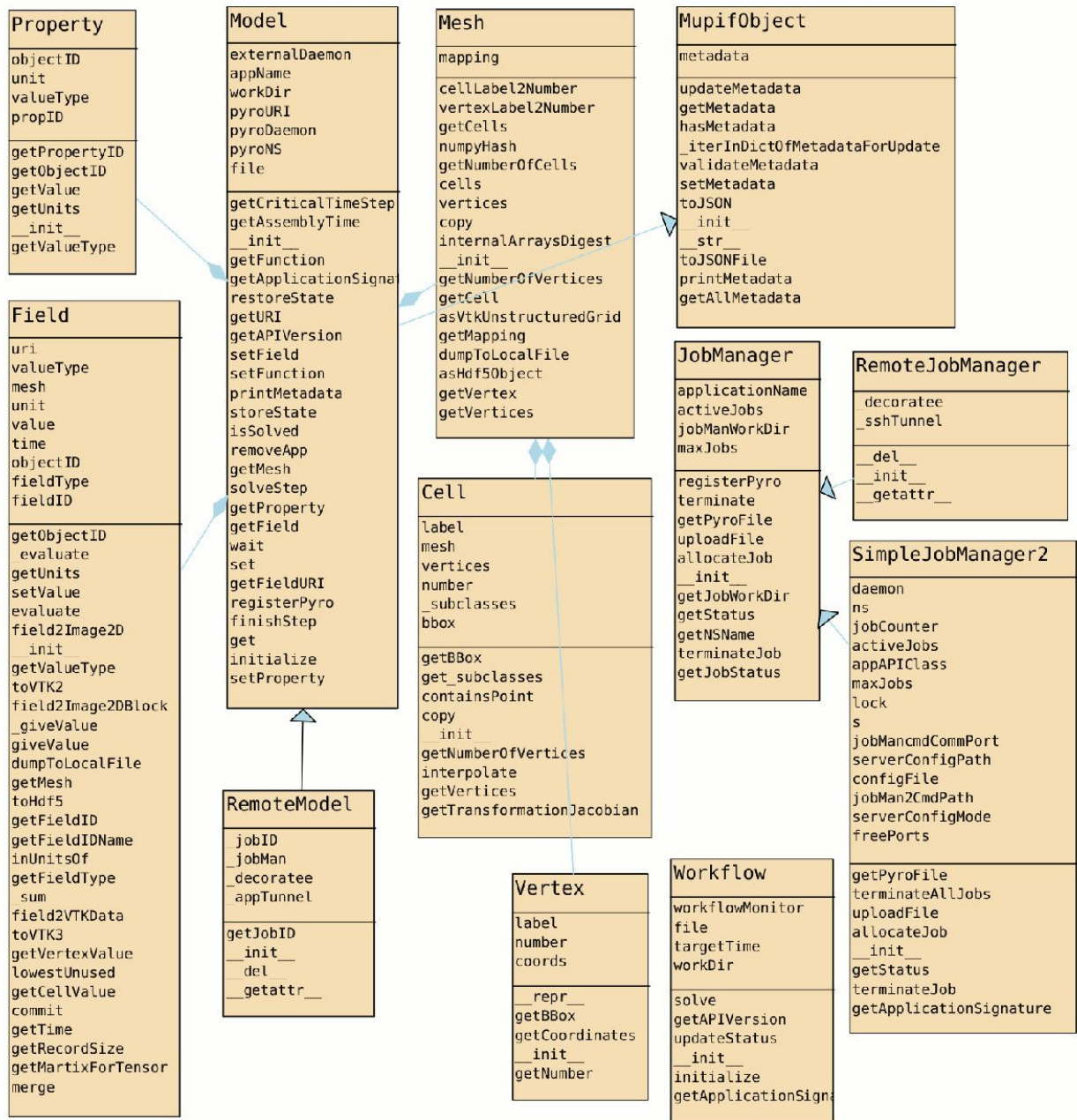


Figure 2: UML diagram of important abstract classes with only selected relations are displayed (Using Pynsource package for UML diagram)

5.1. Common API for all components

The object-oriented approach allows to define hierarchy of classes. This is also used in designing MuPif class structure, where all component classes form a hierarchy, where on top of this hierarchy is *MupifObject* class. This class introduces a common interface that is then

inherited by all derived classes, thus by all MuPIF components involving models (Model class), workflows, and high-level data components, such as properties or spatial fields.

The *MupifObject* class essentially defines methods allowing to get/set metadata to the component. The metadata are identified by unique ID and can be of any type. Internally, they are stored in internal dictionary declared by *MupifObject*.

Service	Description
<code>__init__ (self, jsonFileName='')</code>	<p>Constructor. Initializes the <i>MupifObject</i>.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - <code>jsonFileName</code>: Optionally JSON filename to instantiate from <p>None.</p>
<code>getMetadata (self, key)</code>	<p>Returns metadata associated to given key</p> <p>ARGS:</p> <ul style="list-style-type: none"> - <code>Key</code>: unique metadataID identifying metadata <p>Returns: metadata associated to key</p> <p>Throws <i>TypeError</i> if key does not exist</p>
<code>hasMetadata(self, key)</code>	<p>Tests whether metadata with given key exist</p> <p>Returns: true if metadata entry exist, false otherwise</p>
<code>setMetadata(self, key, val)</code>	<p>Sets metadata associated to key</p> <p>ARGS:</p> <ul style="list-style-type: none"> - <code>Key</code>: unique metadataID identifying metadata - <code>val(any_type)</code>: metadata
<code>getAllMetadata(self):</code>	<p>Gets all metadata of receiver</p> <p>Returns: dict with receiver metadata</p>
<code>printMetadata(self, nonEmpty=False)</code>	<p>Prints receiver metadata</p> <p>ARGS:</p> <ul style="list-style-type: none"> - <code>nonEmpty</code>: if true only keys with nonempty value will be printed
<code>updateMetadata(self, dictionary)</code>	<p>Updates receiver metadata with given metadata</p> <p>ARGS:</p> <ul style="list-style-type: none"> - <code>Dictionary</code>: dict containing metadata to update.

<code>validateMetadata(self, template)</code>	Validates the receiver metadata to given template ARGS: - Template: dict with JSON schema template
<code>toJson(self)</code>	Returns JSON string representation of receiver Returns: JSON representation (string)

5.1.1 Metadata and metadata schemas

The metadata and metadata schemas in MuPIF are stored in a form of JSON representations as a nested (hierarchical) dictionary. JSON stands for “JavaScript Object Notation”, a simple data interchange format. In its heart, JSON is built on the following data structures: object, array, number, string, boolean and null. With these simple data types, all kinds of structured data can be represented. The JSON schema is a template defining what fields are expected, and how the values are represented. The metadata can be validated against schema. The JSON schema itself is written in JSON. The JSON schema standard can be found in [11].

In short, a schema in a Python is represented as a python dictionary, with following keys: *type*, *properties*, and *required*.

- The *type* defines the type of data. Can be any of the supported JSON types (object, array, number, string, boolean or null)
- The *properties* is a dictionary containing the actual metadata in the form of key-value pairs, where values in the schema are dictionaries, containing ‘type’ key defining type of property.
- The *required* key is an array containing required property keys.

```
#Example of model schema (from Model.py)
ModelSchema = {
    'type': 'object',
    'properties': {
        'Name': {'type': 'string'},
        'ID': {'type': ['string', 'integer']},
        'Description': {'type': 'string'},
        'Material': {'type': 'string'},
        'Physics': {
            'type': 'object',
            'properties': {
                'Type': {'type': 'string', 'enum': ['Electronic', 'Atomistic', 'Molecular',
'Continuum', 'Other']},
                'Entity': {'type': 'string', 'enum': ['Atom', 'Electron', 'Grains', 'Finite
volume', 'Other']}
            },
            'required': ['Type', 'Entity']
        },
    },
    'required': ['Name', 'ID', 'Description', 'Physics']
}
```

```
}
```

The following listing shows valid metadata (according to schema defined above):

```
# Example of valid metadata
metadata = {
  'Name': 'Stationary thermal problem',
  'ID': 'Thermo-1',
  'Description': 'Stationary heat conduction using finite elements on rectangular domain',
  'Geometry': '2D rectangle',
  'Physics': {
    'Type': 'Continuum',
    'Entity': 'Finite volume',
    'Equation': ['Heat balance'],
    'Equation_quantities': ['Heat flow'],
    'Relation_description': ['Fick\'s first law'],
    'Relation_formulation': ['Flow induced by thermal gradient on isotropic material'],
    'Representation': 'Finite volumes'
  },
}
```

As illustrated, metadata can contain nested data structures. It is possible to access the individual metadata entries by using convenience methods provided by any *MupifObject* instance. Also, it is possible to insert a new metadata entry to the structure. These methods allow to use ‘dot’ notation to access nested entries, as illustrated in the example below:

```
myobj.getMetadata ('Name')    # returns 'Stationary thermal problem'
myobj.getMetadata ('Physics.Type') #returns 'Continuum'
myobj.setMetadata ('Physics.Representation', 'Finite elements') # change existing entry
myobj.setMetadata ('Physics.NewNote', 'My note') # add a new entry to metadata
```

The metadata schemata are defined in corresponding modules. In MuPIF, the metadata schema is defined for *Model*, *Workflow*, and all data classes (in `dataID.py`).

5.2. Model class

This abstract class represents an external model and defines its interface. The interface is defined in terms of abstract services for data exchange and steering. Derived classes represent individual simulation models. In terms of MODA [9] nomenclature, introduced by EMMC [10], the instances of *Model* class correspond to MODA models and post-processing tools. The *Application* class is a synonym for *Model* class, but is deprecated.

The data exchange services consist of methods for getting and registering external properties, fields, and functions, which are represented using corresponding, newly introduced classes. Steering services allow invoking (execute) solution for a specific solution step, update solution state, terminate the application, etc.

Service	Description
<code>__init__(self, metaData={})</code>	<p>Constructor. Initializes the application. Should define problem independent part of model metadata.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - metaData (dict): metadata to merge with receiver metadata
<code>initialize(self, file='', workdir='', metaData={}, validateMetaData=True, **kwargs)</code>	<p>Initializes application, i.e. should perform all steps after constructor and before run.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - file (str): optional path to input file - workdir (str): optional path to working directory - metaData (dict): metadata to merge with receiver metadata - validateMetaData (bool): if true, receiver metadata will be validated against class metadata template - Kwargs: any optional arguments
<code>registerPyro (self, pyroDaemon, pyroNS, pyroURI, appName=None, externalDaemon=False)</code>	<p>Register the Pyro daemon and nameserver. Required by several services</p> <p>ARGS:</p> <ul style="list-style-type: none"> - pyroDaemon(Pyro4.Daemon): Optional pyro daemon - pyroNS(Pyro4.naming.Nameserver): Optional nameserver - PyroURI (str): Optional URI of receiver - appName (str): Optional application name. Used for removing from pyroNS - externalDaemon (bool): Optional parameter when daemon was allocated externally.
<code>get(self, objectTypeID, time=None, objectID=0)</code>	<p>Returns the requested data object at given time. Object is identified by id.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - objectTypeID (PropertyID or FieldID or FunctionID): Identifier of the object - time (Physics.PhysicalQuantity): target time

	<ul style="list-style-type: none"> - objectID (int): Identifies object with objectID (optional, default 0) Returns: Returns instance of requested data object.
set(self, obj, objectID=0)	Registers the given (remote) data object in model (application). ARGS: <ul style="list-style-type: none"> - obj (Property.Property or Field.Field or Function.Function): data object to register - objectID (int): Identifies object with objectID (optional, default 0)
getField(self, fieldID, time)	Returns the requested field at given time. Field is identified by fieldID. Deprecated, use get method instead. ARGS: <ul style="list-style-type: none"> - fieldID (FieldID): identifier - time (PhysicalQuantity): target time Returns: Returns requested field (Field).
getFieldURI(self, fieldID, time)	Returns the uri of requested field at given time. Field is identified by fieldID. ARGS: <ul style="list-style-type: none"> - fieldID (FieldID): identifier - time (PhysicalQuantity): target time Returns: Returns requested field (Field).
setField(self, field)	Registers the given (remote) field in application. Deprecated, use set method instead. ARGS: <ul style="list-style-type: none"> - field (Field): remote field to be registered by the application Returns: None

getProperty(self, propID, time, objectID=0)	<p>Returns property identified by its ID evaluated at given time. Deprecated, use get() method instead.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - propID (PropertyID): property ID - time (PhysicalQuantity): time when property to be evaluated - objectID (int): identifies object/submesh on which property is evaluated (optional) <p>Returns:</p> <p>Returns representation of requested property (Property).</p>
setProperty(self, property, objectID=0)	<p>Register given property in the application. Deprecated, use set method instead.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - property (Property): the property class - objectID (int): identifies object/submesh on which property is evaluated (optional) <p>Returns:</p> <p>None</p>
getFunction(self, funcID, objectID=0)	<p>Returns function identified by its ID. Deprecated, use get method instead.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - funcID (FunctionID): function ID - objectID (int): identifies optional object/submesh <p>Returns:</p> <p>Returns requested function(Function)</p>
setFunction(self, func, objectID=0)	<p>Register given function in the application. Deprecated use set method instead.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - Func (Function): function to register - objectID (int): identifies optional object/submesh
getMesh (self, timestep)	<p>Returns the computational mesh for given solution step.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - timestep (TimeStep): solution step <p>Returns:</p> <p>Returns the representation of mesh (Mesh)</p>

<code>solveStep(self, timestep, stageID=0, runInBackground=False)</code>	<p>Solves the problem for a given time step. Proceeds the solution from actual state to given time. The actual state should not be updated at the end, as this method could be called multiple times for the same solution step until the global convergence is reached. When global convergence is reached, <code>finishStep</code> is called and then the actual state has to be updated.</p> <p>Solution can be split into individual stages identified by optional <code>stageID</code> parameter. In between the stages the additional data exchange can be performed. See also <code>wait</code> and <code>isSolved</code> services.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - <code>timestep(TimeStep)</code>: solution step - <code>stageID(int)</code>: optional argument identifying solution stage - <code>runInBackground(bool)</code>: if set to <code>True</code>, the solution will run in background (in separate thread), if supported. <p>Returns: None</p>
<code>wait(self)</code>	<p>Wait until solve is completed when executed in background.</p> <p>Returns: None</p>
<code>isSolved(self)</code>	<p>Returns true or false depending whether solve has completed when executed in background.</p> <p>Returns: (Boolean)</p>
<code>finishStep(self, timestep)</code>	<p>Called after a global convergence within a time step.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - <code>timestep(TimeStep)</code>: solution step <p>Returns: None</p>
<code>getCriticalTimeStep(self)</code>	<p>Returns the actual (related to the current state) critical time step increment.</p> <p>Returns: critical time step (PhysicalQuantity)</p>
<code>getAssemblyTime(self, timestep)</code>	<p>Returns the assembly time related to a given time step. The registered fields (inputs) should be evaluated in this time.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - <code>timestep (TimeStep)</code>: solution step <p>Returns: assembly time (PhysicalQuantity)</p>

<code>storeState(self, timestep)</code>	Store the solution state of an application. ARGS: - timestep(TimeStep): solution step Returns: None
<code>restoreState(self, timestep)</code>	Restore the saved state of an application. ARGS: - timestep(TimeStep): solution step Returns: None
<code>getAPIVersion(self)</code>	Returns the supported API version. Returns: API version (str, int)
<code>getApplicationSignature(self)</code>	Get application signature. Returns: Returns the application identification (str)
<code>removeApp(self, nameServer=None, appName=None):</code>	Removes (unregisters) application from the name server. ARGS: - nameServer (Pyro4.naming.Namserver): Optional instance of a nameServer - appName (str): Optional name of the application to be removed Returns: None
<code>terminate(self)</code>	Terminates the application. Shutdowns daemons if created internally. Returns: None
<code>getURI(self)</code>	Returns: Returns the application URI or None if application not registered in Pyro (str)

5.3. Workflow class

This abstract class represents a simulation workflow. Workflow can combine several applications into a complex simulation task. A key feature of *Workflow* class is that it is derived from *Model (Application)* class, so it shares the same API as *Model* Interface. This essentially allows to treat any *Workflow* as *Model* and allows to build a hierarchy of nested workflows. In addition, the following services are declared:

Service	Description
---------	-------------

<code>__init__(self, metaData={})</code>	<p>Constructor. Initializes the workflow. Should define problem independent part of metadata.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - metaData (dict): optional pass metadata to merge
<code>initialize(self, file='', workdir='', targetTime=PQ.PhysicalQuantity(0., 's'), metaData={}, validateMetaData=True, **kwargs)</code>	<p>Initializes the workflow.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - file (str): path to application initialization file. - workdir (str): Optional parameter for working directory - targetTime (Physics.PhysicalQuantity): Optional parameter for target time - metaData (dict): optional metadata to merge - validateMetaData (bool): if true, receiver metadata will be validated against class metadata template
<code>solve(self, runInBackground=False):</code>	<p>Solves the workflow.</p> <p>The default implementation solves the problem in series of time steps using solveStep method (inherited) until the final time is reached.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - runInBackground (bool): optional argument, default False. If True, the solution will run in background (in separate thread or remotely). <p>Returns: None</p>
<code>getAPIVersion(self)</code>	<p>Returns the supported API version.</p> <p>Returns: API version (str, int)</p>
<code>getApplicationSignature(self)</code>	<p>Get application signature.</p> <p>Returns: Returns the application identification (str)</p>
<code>updateStatus(self, status, progress=0)</code>	<p>Updates the workflow status. The status is submitted to workflow monitor, if registered on nameserver.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - status (str): string describing the workflow status (initialized, running, failed, finished) - progress (int): integer number indicating execution progress (in percent)

5.4. Property class

Property is a characteristic value of a problem, which has no spatial variation. Property is identified by *PropertyID*, which is an enumeration determining its physical meaning. It can represent any quantity of a scalar, vector, or tensorial type. Property keeps its value, type, associated time and an optional *objectID*, identifying related component/subdomain.

Service	Description
<code>__init__(self,propID, valueType,units, time=None,objectID=0)</code>	<p>Constructor, initializes the property.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - value (tuple): value of a property. Scalar value is represented as array of size 1. Vector is represented as values packed in a tuple. Tensor is represented as 3D tensor stored in a tuple, column by column. - propID (PropertyID): property ID - valueType (ValueType): type of property value - time (Physics.PhysicalQuantity): time - units (Physics.PhysicalUnits, None): property units or None for time independent property - objectID (int): optional ID of problem object / subdomain to which property is related.
<code>loadFromLocalFile(cls,fileName)</code>	<p>Alternative constructor from a Pickle module.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - filename (str): File name <p>Returns:</p> <p>Property instance (Property)</p>
<code>getValue(self, time=None, *kwargs)</code>	<p>Returns the value of property in a tuple.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - time (Physics.PhysicalQuantity): time to evaluate receiver at. - **kwargs: Arbitrary keyword arguments, see documentation of derived classes <p>Returns:</p> <p>Property value as array (Physics.PhysicalQuantity)</p>
<code>getValueType(self)</code>	<p>Returns the value type of property.</p> <p>Returns:</p> <p>Property value type (mupif.PropertyID)</p>
<code>getPropertID(self)</code>	<p>Returns type of property.</p> <p>Returns:</p> <p>Receiver property ID (PropertyID)</p>

<code>getObjectID(self)</code>	Returns property objectID. Returns: ID of related object (int)
<code>getUnits(self)</code>	Returns representation of property units. Returns: Returns receiver's units (Physics.PhysicalUnits)
<code>dumpToLocalFile(self, fileName, protocol=pickle.HIGHEST_PROTOCOL)</code>	Dump Property to a file using Pickle module ARGS: - filename (str): File name - protocol (int): Used protocol - 0=ASCII, 1=old binary, 2=new binary
<code>inUnitsOf(self, *units)</code>	Express the quantity in different units. If one unit is specified, a new <code>PhysicalQuantity</code> object is returned that expresses the quantity in that unit. If several units are specified, the return value is a tuple of <code>PhysicalObject</code> instances with with one element per unit such that the sum of all quantities in the tuple equals the original quantity and all the values except for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second. ARGS: - Units (C{str}): one units Returns: One physical quantity (L{PhysicalQuantity} or C{tuple}) Raises: TypeError: if any of the specified units are not compatible with the original unit

Property with constant value in time is represented by *ConstantProperty* class derived from *Property* class. In the following, we just mention the modifications in interface from *Property* class.

Service	Description
---------	-------------

<code>__init__(self, value, propID, valueType, units, time=None, objectID=0)</code>	<p>Constructor, initializes the property.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - <code>value (tuple)</code>: value of a property. Scalar value is represented as array of size 1. Vector is represented as values packed in a tuple. Tensor is represented as 3D tensor stored in a tuple, column by column. - <code>propID (PropertyID)</code>: property ID - <code>valueType (ValueType)</code>: type of property value - <code>time (Physics.PhysicalQuantity)</code>: time; None for constant property in time - <code>units (Physics.PhysicalUnits, string)</code>: property units or string - <code>objectID (int)</code>: optional ID of problem object / subdomain to which property is related.
<code>getTime(self)</code>	<p>Returns time associated with this property.</p> <p>Returns:</p> <p>Time (Physics.PhysicalQuantity)</p>

5.5. Field class

Representation of field. *Field* is a scalar, vector, or tensorial quantity defined on a spatial domain (represented by the *Mesh* class). The field provides interpolation services in space, but is assumed to be fixed in time (the application interface allows to request field at specific time). The fields are usually created by the individual applications (sources) and being passed to target applications. The field can be evaluated in any spatial point belonging to underlying domain. Derived classes will implement fields defined on common discretizations, like fields defined on structured or unstructured FE meshes, finite difference grids, etc. Basic services provided by the field class include a method for evaluating the field at any spatial position and a method to support graphical export (creation of VTK dataset).

Service	Description
---------	-------------

<code>__init__(self, mesh, fieldID, valueType, units, time, values=None, fieldType=FieldType.FT_vertexBased, objectID=0, metaData={})</code>	<p>Constructor. Initializes the field instance.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - mesh (Mesh): Instance of Mesh class representing underlying discretization. - fieldID (FieldID): field type - valueType (ValueType): type of field values - units (Physics.PhysicalUnits): Field value units - time (double): time - values (tuple): field values, usually at mesh vertices (format dependent of particular field type) - fieldType (FieldType): Optional, determines field type (values specified as vertex or cell values), default is FT_vertexBased - objectID (int): Optional ID of problem object/subdomain to which field is related - metaData (dict): Optionally pass metadata to merge
<code>loadFromLocalFile(cls, fileName)</code>	<p>Alternative constructor from a Pickle module.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - filename (str): File name <p>Returns:</p> <p>Field instance (Field)</p>
<code>getRecordSize(self)</code>	<p>Return the number of scalars per value, depending on :obj:`valueType` passed when constructing the instance.</p> <p>Returns:</p> <p>number of scalars (1,3,9 respectively for scalar, vector, tensor) (int)</p>
<code>getMesh(self)</code>	<p>Returns representation of underlying discretization.</p> <p>Returns:</p> <p>Reference to associated mesh (Mesh)</p>
<code>getValueType(self)</code>	<p>Returns type of field values (ValueType) of the receiver.</p> <p>Returns:</p> <p>(ValueType)</p>
<code>getFieldID(self)</code>	<p>Returns FieldID, e.g. FID_Displacement, FID_Temperature.</p> <p>Returns:</p> <p>Returns fieldID (FieldID)</p>
<code>getFieldIDName(self)</code>	<p>Returns name of the field.</p> <p>Returns:</p> <p>Returns fieldID name (str)</p>

<code>getFieldType(self)</code>	<p>Returns receiver field type (values specified as vertex or cell values)</p> <p>Returns: Returns fieldTypeID (FieldType)</p>
<code>getTime(self)</code>	<p>Alternative constructor from a Pickle module.</p> <p>ARGS:</p> <p>Returns: Field time (Physics.PhysicalQuantity)</p>
<code>evaluate(self, position, eps=0.001)</code>	<p>Evaluates the receiver at given spatial position.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - position (tuple, list of tuples): 3D position vector or list of position vectors - eps(double): Optional tolerance <p>Returns: Receiver value or list of values evaluated at given position(s) (Physics.PhysicalQuantity).</p>
<code>getVertexValue(self, componentID)</code>	<p>Returns the value associated with a given vertex component</p> <p>ARGS:</p> <ul style="list-style-type: none"> - componentID (tuple): A tuple identifying a component: vertex (vertexID,) or integration point (CellID, IPID) <p>Returns: component value as (Physics.PhysicalQuantity)</p>
<code>getCellValue(self, componentID)</code>	<p>Returns the value associated with a given integration point on a cell.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - componentID (tuple): A tuple identifying a component: vertex (vertexID,) or integration point (CellID, IPID) <p>Returns: component value as (Physics.PhysicalQuantity)</p>
<code>setValue(self, componentID, value)</code>	<p>Sets the value associated to given component (vertex or cell IP). Note, that the field values are updated after a commit method is invoked.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - componentID (tuple): The componentID is a tuple: (vertexID) or (CellID, IPID) - value(tuple): Component value <p>Returns: None</p>

<code>giveValue(self, componentID):</code>	<p>Returns the value associated with a given component (vertex or integration point on a cell).</p> <p>ARGS:</p> <ul style="list-style-type: none"> - <code>componentID (tuple)</code>: A tuple identifying a component: vertex (vertexID,) or integration point (CellID, IPID) <p>Returns:</p> <p>Tuple of values (tuple)</p>
<code>setValue(self, componentID, value):</code>	<p>Sets the value associated with a given component (vertex or integration point on a cell).</p> <p>ARGS:</p> <ul style="list-style-type: none"> - <code>componentID (tuple)</code>: A tuple identifying a component: vertex (vertexID,) or integration point (CellID, IPID) - <code>value (tuple)</code>: Value to be set for a given component, should have the same units as receiver
<code>commit(self)</code>	<p>Commits the recorded changes (via <code>setValue</code> method).</p> <p>Returns: None</p>
<code>getUnits(self)</code>	<p>Returns:</p> <p>Returns units of the receiver (<code>Physics.PhysicalUnits</code>)</p>
<code>merge(self, field)</code>	<p>Merges the receiver with a given field together. Both fields should be on different parts of the domain (can also overlap), but should be of the same type and refer to the same underlying discretization.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - <code>field (Field)</code>: field to merge <p>Returns: None</p>
<code>field2VTKData (self name=None,lookupTable=None)</code>	<p>Returns VTK representation of the receiver.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - <code>Name (str)</code>: human-readable name of the field - <code>lookupTable (pyvtk.LookupTable)</code>: color lookup table <p>Returns:</p> <p>VTK dataset (pyvtk)</p>

<pre>getMartixForTensor(self,values)</pre>	<p>Reshape values to a list with 3x3 arrays. Usable for VTK export.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - Values (list): List containing tuples of 9 values, e.g. [(1,2,3,4,5,6,7,8,9), (1,2,3,4,5,6,7,8,9), ...] <p>Returns:</p> <p>List containing 3x3 matrices for each tensor (list)</p>
<pre>dumpToLocalFile(self, fileName, protocol=pickle.HIGHEST_PROTOCOL)</pre>	<p>Dump Field to a file using a Pickle serialization module.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - filename (str): File name - protocol (int): Used protocol - 0=ASCII, 1=old binary, 2=new binary
<pre>field2Image2D(self, plane='xy', elevation = (-1.e-6, 1.e-6), numX=10, numY=20, interp='linear', fieldComponent=0, vertex=True, colorBar='horizontal', colorBarLegend='', barRange=(None,None), barFormatNum='%.3g', title='', xlabel='', ylabel='', fileName='', show=True, figsize = (8,4), matPlotFig=None)</pre>	<p>Plots and/or saves 2D image using a matplotlib library. Works for structured and unstructured 2D/3D fields. 2D/3D fields need to define plane. This method gives only basic viewing options, for aesthetic and more elaborated output use e.g. VTK field export with postprocessors such as ParaView or Mayavi.</p> <p>ARGS: see the reference manual</p> <p>Returns: handle to matPlotFig (matPlotFig)</p>
<pre>field2Image2DBlock(self)</pre>	<p>Block an open window from matPlotLib. Waits until closed.</p>
<pre>toHdf5(self,fileName,group='component1/part1')</pre>	<p>Dump field to HDF5, in a simple format suitable for interoperability (TODO: document).</p> <p>ARGS:</p> <ul style="list-style-type: none"> - filename (str): HDF5 file
<pre>makeFromHdf5(fileName,group='component1/part1')</pre>	<p>Restore Fields from HDF5 file.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - filename (str): HDF5 file <p>Returns:</p> <p>list of new :obj:`Field` instances (Field, Field,...)</p>

toVTK2(self,fileName,format='ascii')	<p>Save the instance as Unstructured Grid in VTK2 format (``.vtk``).</p> <p>ARGS:</p> <ul style="list-style-type: none"> - filename (str): where to save - format (str): one of ``ascii`` or ``binary``
makeFromVTK2(fileName,unit,time=0,skip=['coolwarm'])	<p>Return fields stored in *fileName* in the VTK2 (``.vtk``) format.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - filename (str): filename to load from - unit (PhysicalUnit): physical unit of filed values - time (float): time value for created fields (time is not saved in VTK2, thus cannot be recovered) - skip (str,...): file names to be skipped when reading the input file; the default value skips the default coolwarm colormap. <p>Returns: one field from VTK (Field)</p>
toVTK3(self,fileName,**kw)	<p>Save the instance as Unstructured Grid in VTK3 format (``.vtu``). This is a simple proxy for calling :obj:`manyToVTK3` with the instance as the only field to be saved. If multiple fields with identical mesh are to be saved in VTK3, use :obj:`manyToVTK3` directly.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - filename (str): output file name - **kw ():passed to :obj:`manyToVTK3`
manyToVTK3(fields,fileName,ascii=False,compress=True)	<p>Save all fields passed as argument into VTK3 Unstructured Grid file (``*.vtu``). All *fields* must be defined on the same mesh object; exception will be raised if this is not the case.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - filename (str): output file name - ascii (bool): write numbers are ASCII in the XML-based VTU file (rather than base64-encoded binary in XML) - compress (bool): apply compression to the data

makeFromVTK3(fileName,units, time=0,forceVersion2=False)	<p>Create fields from a VTK unstructured grid file (``.vtu``, format version 3, or ``.vtp`` with *forceVersion2*); the mesh is shared between fields.</p> <p>``vtk.vtkXMLGenericDataObjectReader`` is used to open the file (unless *forceVersion2* is set), but it is checked that contained dataset is a ``vtk.vtkUnstructuredGrid`` and an error is raised if not.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - filename (str): filename to load from - unit (Physics.PhysicalUnit): units of read values - time (float): time value for created fields (time is not saved in VTK3, thus cannot be recovered) - forceVersion2 (bool): if ``True``, ``vtk.vtkGenericDataObjectReader`` (for VTK version 2) will be used to open the file, instead of ``vtk.vtkXMLGenericDataObjectReader``; this also supposes *fileName* ends with ``.vtk`` (not checked, but may cause an error). <p>Returns:</p> <p>list of new :obj:`Field` instances (Field,Field,...)</p>
inUnitsOf(self, *units)	<p>Should return a new instance. As deep copy is expensive, this operation should be avoided. Better to use convertToUnits method performing in place conversion.</p>

5.6. Function class

Represents a user defined function. Function is an object defined by mathematical expression and can be a function of spatial position, time, and other variables. Derived classes should implement evaluate service by providing a corresponding expression. The function arguments are packed into a dictionary, consisting of pairs (called items) of keys and their corresponding values.

Service	Description
<code>__init__(self,funcID, objectID=0)</code>	<p>Constructor. Initializes the function.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - funcID (FunctionID): function ID - objectID (int): optional ID of associated subdomain.
<code>evaluate (self, d)</code>	<p>Evaluates the function for given parameters packed as a dictionary. A dictionary is container type that can store any number of Python objects, including other container types.</p>

	<p>Dictionaries consist of pairs (called items) of keys and their corresponding values.</p> <p>Example: <code>d={'x':(1,2,3), 't':0.005}</code> initializes dictionary containing tuple (vector) under 'x' key, double value 0.005 under 't' key.</p> <p>Some common keys:</p> <ul style="list-style-type: none"> - 'x': position vector - 't': time <p>ARGS:</p> <ul style="list-style-type: none"> - d (dictionary): dictionary containing function arguments (number and type depends on particular function) <p>RETURNS: function value (tuple) evaluated for given parameters</p>
<code>getID (self)</code>	<p>Returns receiver's ID.</p> <p>Returns: id (FunctionID)</p>
<code>getObjectID(self)</code>	<p>Returns: returns receiver's object ID (int)</p>

5.7. TimeStep class

Class representing solution time step. The time step manages its number, target time, and time increment.

Service	Description
<code>__init__(self, t, dt, targetTime, units=None, n=1)</code>	<p>Constructor. Initializes the new time step.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - t (double, Physics.PhysicalUnit): time - dt (double, Physics.PhysicalUnit): step length (time increment) - targetTime (double, Physics.PhysicalUnit): target simulation time, type depends on 'units' - units (Physics.PhysicalUnit): optional units for t,dt,targetTime if given as float values - n (int): time step number
<code>getTime(self)</code>	<p>Returns:</p> <p>time step time (Physics.PhysicalQuantity)</p>
<code>getTimeIncrement(self)</code>	<p>Returns:</p> <p>time increment (Physics.PhysicalQuantity)</p>
<code>getTargetTime (self)</code>	<p>Returns:</p> <p>Target time (Physics.PhysicalQuantity)</p>

getNumber(self)	Returns: receiver's number (int)
-----------------	--------------------------------------------

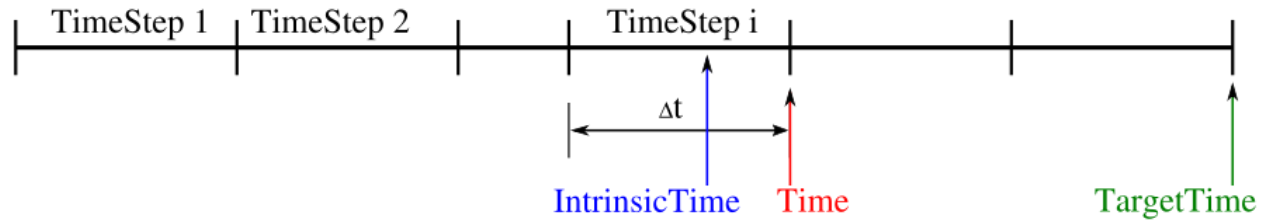


Figure 3: Concept of time step in MuPIF

5.8. Mesh class

Mesh class is an abstract representation of a computational domain and its spatial discretization. The mesh geometry is described using computational cells (representing finite elements, finite difference stencils, etc.) and vertices (defining cell geometry). Derived classes represent structured, unstructured FE grids, FV grids, etc. Mesh is assumed to provide a suitable instance of cell and vertex localizers. In general, the mesh services provide different ways how to access the underlying interpolation cells and vertices, based on their numbers, or spatial location.

Service	Description
<code>__init__(self)</code>	Constructor, creates an empty mesh.
<code>copy(self)</code>	This will return a copy of the receiver. Note, that DeepCopy will not work, as individual cells contain mesh link attributes, leading to underlying mesh duplication in every cell. Returns: Copy of receiver (Mesh)
<code>getNumberOfVertices(self)</code>	Returns: Number of Vertices (int)
<code>getNumberOfCells(self)</code>	Returns: Number of Cells
<code>getVertex(self, i)</code>	Returns i-th vertex (i corresponds to a vertex number, not a label). Returns: vertex (Vertex)

<code>getCell(self, i)</code>	Returns i-th cell (identified by cell number, not label). Returns: cell (Cell)
<code>vertexLabel2Number(self, label)</code>	Returns local vertex number corresponding to given label. If no label corresponds, throws an exception. Returns: vertex number (int)
<code>cellLabel2Number(self, label)</code>	Returns local cell number corresponding to a given label. If no label corresponds, it throws an exception. Returns: cell number (int)
<code>getVerticesInBBox (self, bbox):</code>	Returns the list of all vertices which are inside given bounding Box ARGS: - <code>bbox (BoundingBox):</code> bounding box Returns: list of vertices inside bbox (list)
<code>getCellsInBBox (self, bbox):</code>	Returns the list of cells which bbox intersects with given bounding box ARGS: - <code>bbox (BoundingBox):</code> bounding box Returns: list of cells at least partially in bbox (list)
<code>evaluateVertices(self, functor):</code>	Returns the list of all vertices for which the functor is satisfied. The functor is a user defined class with two methods: <code>giveBBox()</code> which returns an initial functor bbox, and <code>evaluate (obj)</code> which should return true if functor is satisfied for a given object. ARGS: - <code>functor:</code> functor class Returns: list of all vertices for which the functor is satisfied (list)
<code>evaluateCells(self, functor):</code>	Returns the list of all cells for which the functor is satisfied. The functor is user defined class with two methods: <code>getBBox()</code> which returns an initial functor bbox, and <code>evaluate (obj)</code> which should return true if functor is satisfied for given object. ARGS: - <code>functor:</code> functor class Returns: List of all cells for which the functor is satisfied (list)

5.9. Cell class

Representation of a computational cell (finite element). The solution domain is composed of cells, whose geometry is defined using vertices. Cells provide interpolation over their associated

volume, based on given vertex values. Derived classes will be implemented to support common interpolation cells (finite elements, FD stencils, etc.)

Service	Description
<code>__init__(self, mesh, number, label, vertices)</code>	Constructor. Creates the new cell. ARGS: <ul style="list-style-type: none"> - <code>mesh(Mesh)</code>: the mesh to which cell belongs. - <code>number(int)</code>: local cell number - <code>label(int)</code>: cell label - <code>vertices(tuple)</code>: cell vertices (local numbers)
<code>copy(self)</code>	This will copy the receiver, making deep copy of all attributes EXCEPT mesh attribute Returns: the copy of receiver (Cell)
<code>getVertices(self)</code>	Returns: the list of cell vertices (tuple of Vertex instances)
<code>containsPoint(self, point)</code>	Returns: True if cell contains given point, False otherwise
<code>getGeometryType(self)</code>	Returns: geometry type of receiver (CellGeometryType)
<code>getBBox(self)</code>	Returns: bounding box of the receiver (BBox)

5.10. Vertex class

Represents a vertex. In general, a set of vertices defines the geometry of interpolation cells. A vertex is characterized by its position, number and label. Vertex number is locally assigned number (by *Mesh* class), while a label is a unique number defined by application.

Service	Description
<code>__init__(self, number, label, coords=None)</code>	Constructor. Creates the new vertex instance. ARGS: <ul style="list-style-type: none"> - <code>number(int)</code>: local vertex number - <code>label(int)</code>: vertex label - <code>coords(tuple)</code>: 3D position vector of vertex
<code>getCoordinates(self)</code>	Returns: receiver coordinates (tuple)
<code>getNumber(self)</code>	Returns: receiver number (int)

<code>getLabel(self)</code>	Returns: receiver label (int)
-----------------------------	--------------------------------------

5.11. BoundingBox

Represents an axis aligned bounding box - a rectangle in 2d and a prism in 3d. Its geometry is described using two points - lower left and upper right. The bounding box class provides fast and efficient methods for testing whether point is inside and whether an intersection with another bounding box exists.

Service	Description
<code>__init__(self, coords_ll, coords_ur)</code>	<p>Constructor. Creates the new Bounding box instance.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - <code>coords_ll</code> (tuple): coordinates of lower left corner - <code>coords_ur</code> (tuple): coordinates of upper right corner
<code>containsPoint (self, point)</code>	<p>Returns true if point inside receiver.</p> <p>ARGS:</p> <ul style="list-style-type: none"> - <code>point</code> (tuple): point coordinates <p>Returns: True if point is inside receiver, false otherwise (Bool)</p>
<code>intersects (self, bbox)</code>	<p>Returns: Returns true if receiver intersects given bounding box (Bool)</p>
<code>merge (self, entity)</code>	<p>Merges (expands) receiver with given entity (position or bbox)</p> <p>ARGS:</p> <ul style="list-style-type: none"> - <code>entity</code> (tuple or BoundingBox): position vector (tuple) or bounding box. <p>Returns: None</p>

5.12. APIError

This class serves as a base class for exceptions thrown by the framework. Raising an exception is a way to signal that a routine could not execute normally - for example, when an input argument is invalid (e.g. value is outside of the domain of a function) or when a resource is unavailable (like a missing file, a hard disk error, or out-of-memory errors). A hierarchy of specialized exceptions can be developed, derived from the *APIError* class.

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called handlers. To catch exceptions, a portion of code is placed under exception inspection. This is done by enclosing that portion of code in a try-block. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

An exception is thrown by using the throw keyword from inside the try-block. Exception handlers are declared with the keyword "except", which must be placed immediately after the try block.

Service	Description
<code>__init__(self,msg)</code>	Constructor. Initializes the exception. ARGS: <ul style="list-style-type: none">- msg (string) Error message
<code>__str__(self)</code>	Returns: string representation of the exception, ie. error message (string).

6. Developing Application Program Interface (API)

In order to establish an interface between the platform and external application, one has to implement a *Model* class. This class defines a generic interface in terms of general purpose, problem independent, methods that are designed to steer and communicate with the application. The Table 2 presents an overview of application interface, the full details with complete specification can be found in [5.1. Model class](#) specification.

Method	Description
<code>__init__(self, metaData)</code>	Constructor. Initializes the application.
<code>Initialize (file, workdir, metaData, **kwargs)</code>	Initialize model, e.g. set input file, set path
<code>getMesh (self, timestep)</code>	Returns the computational mesh for given solution step.
<code>getField(self, fieldID, time)</code>	Returns the requested field at given time. Field is identified by fieldID.
<code>setField(field)</code>	Registers the given (remote) field in application.
<code>getProperty(self, propID, time, objectID=0)</code>	Returns property identified by its ID evaluated at given time.
<code>setProperty(self, property, objectID=0)</code>	Register given property in the application
<code>setFunction(self, func, objectID=0)</code>	Register given function in the application
<code>solveStep(self, timestep)</code>	Solves the problem for given time step.
<code>finishStep(self, timestep)</code>	Called after a global convergence within a time step.
<code>getCriticalTimeStep()</code>	Returns the actual critical time step increment.
<code>getAssemblyTime(tStep)</code>	Returns assembly time within a timestep
<code>getApplicationSignature()</code>	Returns the application identification
<code>terminate()</code>	Terminates the application.

Table 2: Model interface: an overview of basic methods.

From the perspective of individual simulation tool, the interface implementation can be achieved by means of either direct (native) or indirect implementation.

- **Native implementation** requires a simulation tool written in Python, or a tool with Python interface. In this case the Model services will be implemented directly using direct calls to suitable application's functions and procedures, including necessary internal data conversions. In general, each application (in the form of a dynamically linked library) can be loaded and called, but care must be taken to convert Python data types into target application data types. More convenient is to use a wrapping tool (such as Swig [5] or Boost [6]) that can generate a Python interface to the application,

generally taking care of data conversions for the basic types. The result of wrapping is a set of Python functions or classes, representing their application counterparts. The user calls an automatically generated Python function which performs data conversion and calls the corresponding native equivalent.

- **Indirect implementation** is based on wrapper class implementing Model interface that implements the interface indirectly, using, for example, simulation tool scripting or I/O capabilities. In this case the application is typically standalone application, executed by the wrapper in each solution step. For the typical solution step, the wrapper class has to cache all input data internally (by overloading corresponding set methods), execute the application from previously stored state, passing input data, and parsing its output(s) to collect return data (requested using get methods).

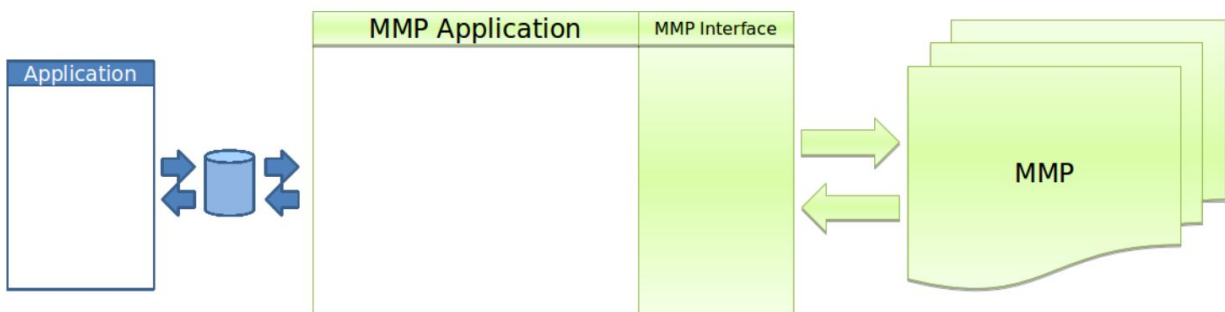


Figure 4: Illustration of indirect approach

The example illustrating the indirect implementation is discussed further. Typically, this is a three-phase procedure. In the first step, when external properties and fields are being set, the application interface has to remember all these values. In the second step, when the application is to be executed, the input file is to be modified to include the mapped values. After the input file(s) are generated, the application itself is executed. In the last, third step, the computed properties/fields are requested. They are typically obtained by parsing application output and returned.

In this example, the application should compute the average value from mapped values of concentrations over the time. The external application is available, that can compute an average value from the input values given in a file. The application interface accumulates the mapped values of concentrations in a list data structure, this is done is setProperty method. During the solution step in a solveStep method, the accumulated values of concentrations over the time are written into a file, the external application is invoked taking the created file as input and producing an output file containing the computed average. The output file is parsed when the average value is requested using getProperty method.

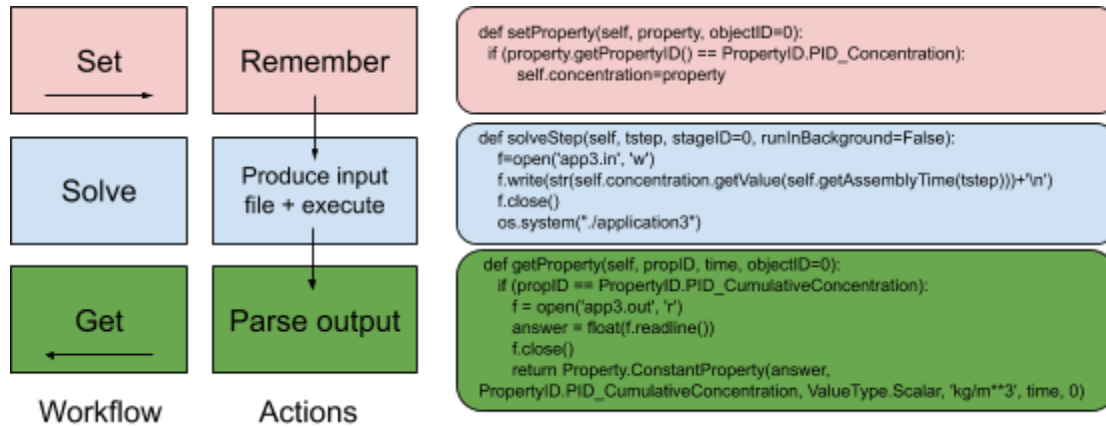


Figure 5: Typical workflow in indirect approach for API implementation

7. Developing user workflows

Multiscale/multiphysics simulations are natively supported in MuPIF, allowing easy data passing from one model to another one, synchronizing and steering all models. Simulation workflow of multiscale/multiphysics simulations, called also a simulation scenario, defines data flow among various models and their steering. Natively, the workflow in MuPIF is represented as Python script combining MuPIF components into workflow. However, a many benefits can be further gained by implementing a workflow as class derived from abstract *Workflow* class. The benefits and example are discussed in chapter [“Workflow as a class”](#).

7.1. Workflow templates

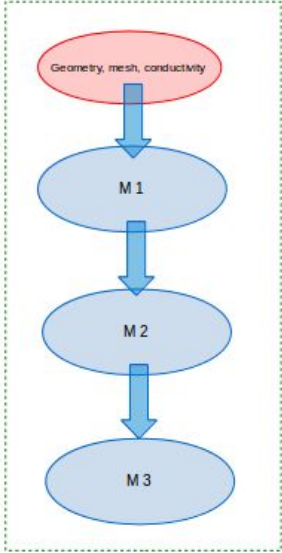
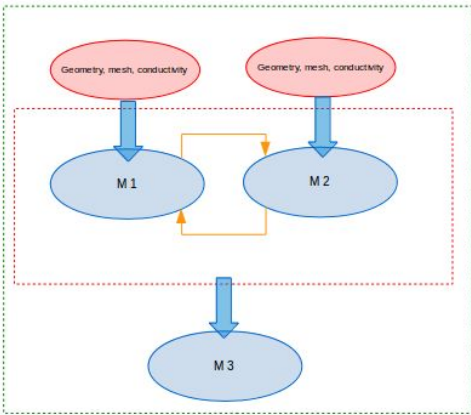
	<pre> time = PQ.PhysicalQuantity('0 s') timeStepNumber = 0 targetTime = PQ.PhysicalQuantity('10 s') while (abs(time-targetTime).getValue() > 1.e-6): dt=min(m1.getCriticalTimeStep(), m2.getCriticalStep(), m3.getCriticalStep()) time = time+dt if (time>targetTime): time=targetTime timeStepNumber = timeStepNumber+1 istep=TimeStep.TimeStep(time, td, targetTime, n=timeStepNumber) try: m1.solveStep(istep) p = m1.getProperty(PID, m2.getAssemblyTime(istep)) m2.setProperty(p) m2.solveStep(istep) ... m3.solveStep(istep) except APIError.APIError as e: print ("Appi Error occurred:",e) break m1.terminate() m2.terminate() m3.terminate() </pre>
-----------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Table 3: Sequential workflow template

	<pre> time = PQ.PhysicalQuantity('0 s') timeStepNumber = 0 targetTime = PQ.PhysicalQuantity('10 s') while (abs(time-targetTime).getValue() > 1.e-6): dt=min(m1.getCriticalTimeStep(), m2.getCriticalStep(), m3.getCriticalStep()) time = time+dt if (time>targetTime): time=targetTime timeStepNumber = timeStepNumber+1 istep=TimeStep.TimeStep(time, td, targetTime, n=timestep) try: convergedFlag = False while not coveredFlag: m1.solveStep(istep) p1=m1.getProperty(id, m2.getAssemblyTime(istep)) m2.setProperty(p1) m2.solveStep(istep) p2=m2.getProperty(id2, m1.getAssemblyTime(istep)) m1.setProperty(p2) #check for convergence convergedFlag=checkConvergence() m3.solveStep() except APIError.APIError as e: print ("Appi Error occurred:",e) break m1.terminate() m2.terminate() m3.terminate() </pre>
Table 4: Loosely coupled workflow template	

7.2. Workflow example

A thermo-mechanical, multiphysical example *Example06.py* explains linking and steering in greater detail. The example presents a local (non-distributed) version and can be found under *examples/Example06** directory of MuPIF installation.

A cantilever, clamped on the left hand side edge, is subjected to stationary temperature loading, see Figure 6. Heat convection is prescribed on the top edge with ambient temperature 10°C. Left and bottom edges have prescribed temperature 0°C, the right edge has no boundary

condition. Initial temperature is set to 0°C , heat conductivity is 1 W/m/K , heat capacity 1.0 J/kg/K , material density 1.0 kg/m^3 . The material has assigned Young's modulus as 30 GPa , Poisson's ratio 0.25 and coefficient of linear thermal expansion $12\text{e-}6^{\circ}\text{C}^{-1}$.

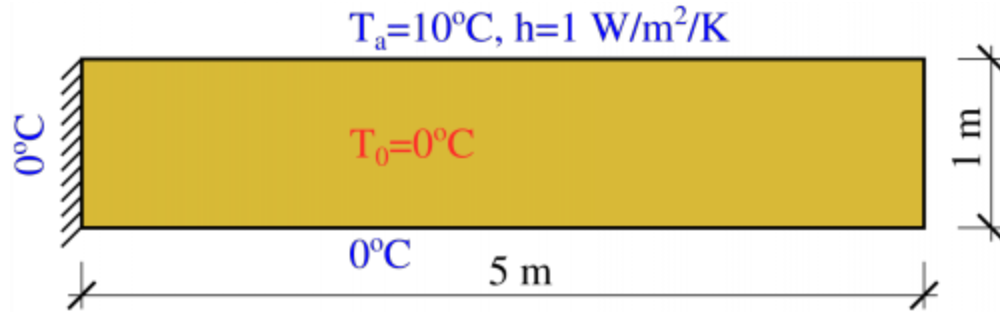


Figure 6: Elastic cantilever subjected to thermal boundary conditions.

First, the temperature distribution has to be solved in the whole domain from the given initial and boundary conditions. The temperature field is passed afterwards to the mechanical analysis, which evaluates the corresponding displacement field. Such simulation flow is depicted in Figure 7, linking two models in one time step. The thermal model implements *getField(T)* and *solveStep(istep)* methods. In addition, the mechanical model needs to set up an initial thermal field *setField(T)* prior to execution in the time step.

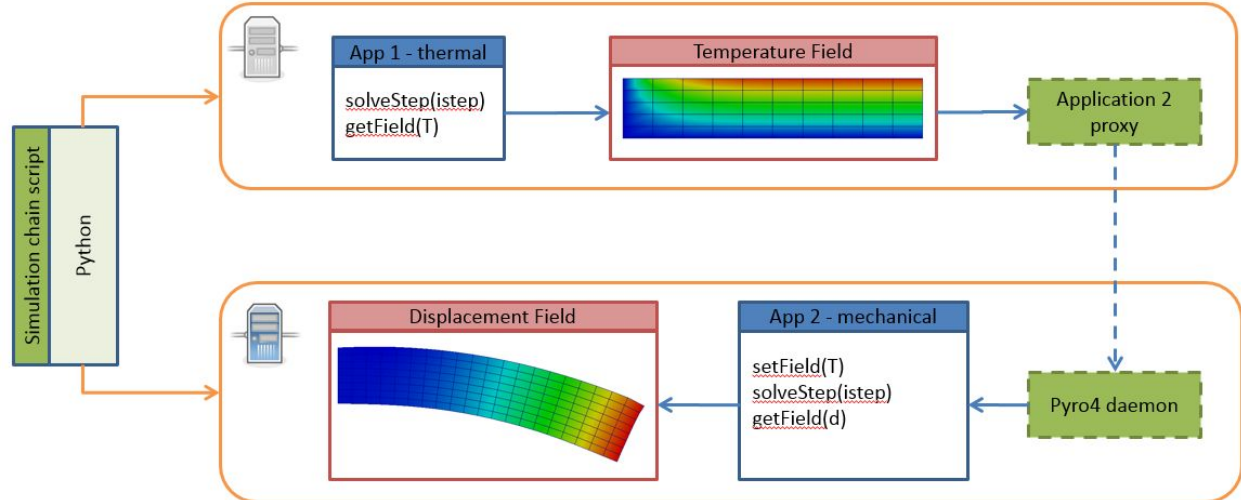


Figure 7: Thermo-mechanical simulation flow

The discretizations for thermal and mechanical problems are in this particular case different and the platform takes care of field interpolation. The mesh for thermal problem consist of 50 linear elements with linear approximation and 55 nodes. The mesh for mechanical analysis consist of 168 nodes and 160 elements with linear approximation. Results for final step are shown in Figure 8.

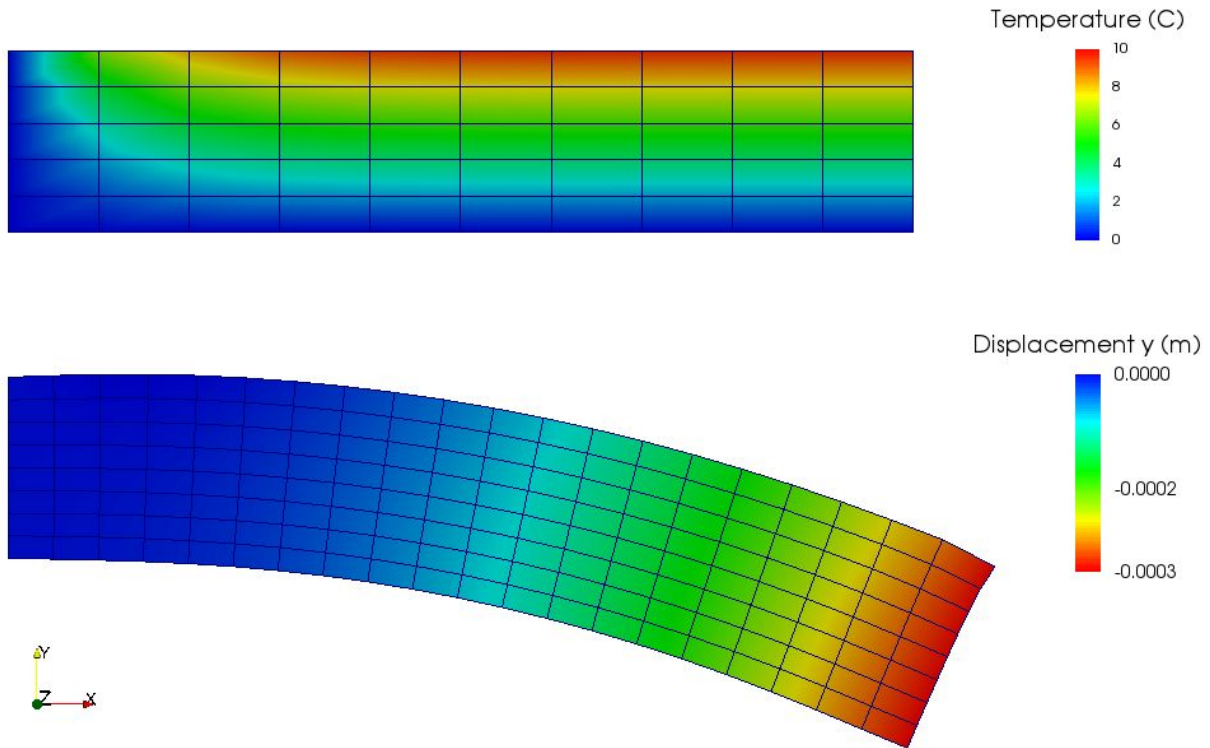


Figure 8: Results of thermo-mechanical simulation

A code below shows a thermo-mechanical simulation in *Example06*. Thermal and mechanical solvers are implemented as *demoapp* module and loaded.

```
class Example06(Workflow.Workflow):
    def __init__(self, metaData={}):
        MD = {
            'Name': 'Thermo-mechanical stationary problem',
            'ID': 'Thermo-mechanical-1',
            ...
        }
        super(Example06, self).__init__(metaData=MD)
        self.updateMetadadata(metaData)

        self.thermalSolver = demoapp.thermal()
        self.mechanicalSolver = demoapp.mechanical()

    def initialize(self, file='', workdir='', targetTime=PQ.PhysicalQuantity('0 s'),
        metaData={}, validateMetaData=True, **kwargs):
        super(Example06, self).initialize(file=file, workdir=workdir, targetTime=targetTime,
        metaData=metaData, validateMetaData=validateMetaData, **kwargs)

        passingMD = { ... }
```

```

        self.thermalSolver.initialize('inputT10.in', '.', metaData=passingMD)
        self.mechanicalSolver.initialize('inputM10.in', '.', metaData=passingMD)

    def solveStep(self, istep, stageID=0, runInBackground=False):
        self.thermalSolver.solveStep(istep, stageID, runInBackground)
        self.mechanicalSolver.setField(self.thermalSolver.getField(FieldID.FID_Temperature,
istep.getTime()))
        self.mechanicalSolver.solveStep(istep, stageID, runInBackground)

    def getField(self, fieldID, time, objectID=0):
        if fieldID == FieldID.FID_Temperature:
            return self.thermalSolver.getField(fieldID, time, objectID)
        elif fieldID == FieldID.FID_Displacement:
            return self.mechanicalSolver.getField(fieldID, time, objectID)
        else:
            raise APIError.APIError('Unknown field ID')

    def getCriticalTimeStep(self):
        return PQ.PhysicalQuantity(1.0, 's')

    def terminate(self):
        self.thermalSolver.terminate()
        self.mechanicalSolver.terminate()
        super(Example06, self).terminate()

    def getApplicationSignature(self):
        return "Example06 workflow 1.0"

    def getAPIVersion(self):
        return "1.0"

md = {
    'Execution': {
        'ID': '1',
        'Use_case_ID': '1_1',
        'Task_ID': '1'
    }
}

demo = Example06()
demo.initialize(targetTime=PQ.PhysicalQuantity('1 s'), metaData=md)

tstep = TimeStep.TimeStep(
    PQ.PhysicalQuantity('1 s'),
    PQ.PhysicalQuantity('1 s'),
    PQ.PhysicalQuantity('10 s')
)

demo.solveStep(tstep)
demo.terminate()

```

Listing 2: *Example13* showing a thermo-mechanical simulation

As already mentioned, the thermo-mechanical simulation chain can run in various configurations, composed of a steering script, nameserver, thermal and mechanical

applications, using ssh or VPN network connection. Table 3 shows MuPIF examples of thermo-mechanical configuration. In principle, each component can run on different computer, except a steering script.

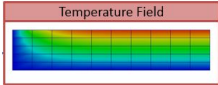
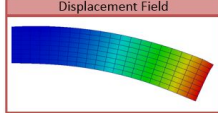
	Steering script	Nameserver	Thermal application 	Mechanical application 
Example06 (local)	Local	-	Local	Local
Example07 (JobMan,VPN, ssh)	Local	Remote	Remote, JobMan	Remote, JobMan
Example08 (JobMan, VPN, ssh)	Local	Remote	Remote, JobMan	Local

Table 3: Examples of thermo-mechanical simulation on local and various distributed configurations.

7.3. Workflow as a class

The object oriented design of MuPIF allows to build a hierarchy of workflows, where the top level workflow may utilise the components, which may be again workflows. From this point of view, any workflow can be regarded as an application, composed from individual components, implementing itself an application interface. The application interface, as introduced in Chapter on Platform APIs, allows to perform any data and steering operation, i.e. to get and set any data, update response for the given solution step, etc.

Another important advantage of having workflow represented as a class is that the individual workflows can be allocated and executed by a jobManager on remote resources in a same way as individual applications.

MuPIF comes with abstract *Workflow* class, derived from *Model* class, supposed to be a parent class for any workflow represented as a class. It extends the *Model* interface by defining *solve* method, which implements a time loop over the individual time steps, solved by *solveStep* method defined already in *Model* interface.

The default implementation of *Workflow*'s *solve* method is shown in a listing below. It generates a sequence of time steps satisfying the stability requirements till reaching the target time. If the default implementation does not fit, the method can be overloaded.

```
class Workflow(Model.Model):
    def solve(self, runInBackground=False):
```

```

self.setMetadata('Status', 'Running')
self.setMetadata('Progress', 0.)
time = PQ.PhysicalQuantity('0.0 s')
timeStepNumber = 0

while (abs(time.inUnitsOf(timeUnits).getValue()-
        self.targetTime.inUnitsOf(timeUnits).getValue()) > 1.e-6):
    dt = self.getCriticalTimeStep()
    time=time+dt
    if (time > self.targetTime):
        time = self.targetTime
    timeStepNumber = timeStepNumber+1
    istep=TimeStep.TimeStep(time, dt, self.targetTime, n=timeStepNumber)

    log.debug("Step %g: t=%g dt=%g"% (timeStepNumber,
        time.inUnitsOf(timeUnits).getValue(),
        dt.inUnitsOf(timeUnits).getValue()))

    self.solveStep(istep)
    self.finishStep(istep)
self.setMetadata('Status', 'Finished')
self.setMetadata('Date_time_end', timeTime.strftime("%Y-%m-%d %H:%M:%S",
timeTime.gmtime()))

self.terminate()

```

8. Distributed Model

Common feature of parallel and distributed environments is a distributed data structure and concurrent processing on distributed processing nodes. This brings in an additional level of complexity that needs to be addressed. To facilitate execution and development of the simulation workflows, the platform provides the transparent communication mechanism that will take care of the network communication between the objects. An important feature is the transparency, which hides the details of remote communication to the user and allows to work with local and remote objects in the same way.

The communication layer is built on [Pyro library](#) [4], which provides a transparent distributed object system fully integrated into Python. It takes care of the network communication between the objects when they are distributed over different machines on the network. One just calls a method on a remote object as if it were a local object – the use of remote objects is (almost) transparent. This is achieved by the introduction of so-called proxies. A proxy is a special kind of object that acts as if it were the actual object. Proxies forward the calls to the remote objects, and pass the results back to the calling code. In this way, there is no difference between

simulation script for local or distributed case, except for the initialization, where, instead of creating local object, one has to connect to the remote object.

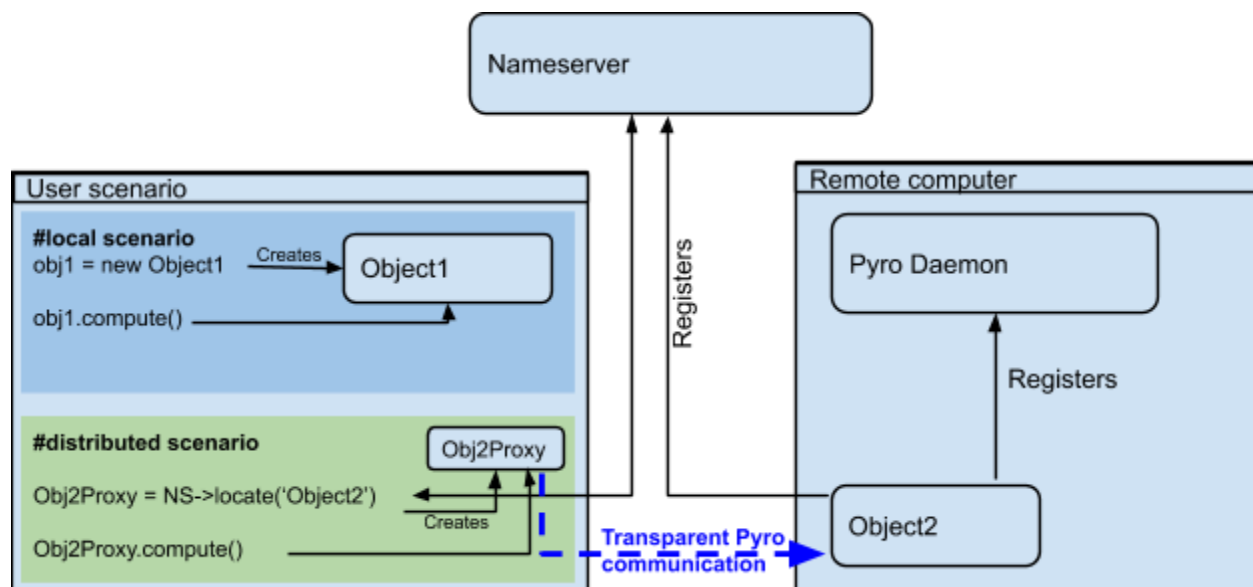


Figure 9: Comparison of local vs. remote object communication scenarios

To make an object remotely accessible, it has to be registered with the daemon, a special object containing server side logic which dispatches incoming remote method calls to the appropriate objects. To enable runtime discovery of the registered objects, the name server is provided, offering a phone book for Pyro objects, allowing to search for objects based on logical name. The name server provides a mapping between logical name and exact location of the object in the network, so called uniform resource identifier (URI). The process of object registration and of communication with remote objects (compared to local objects) is illustrated in Fig. 9.

8.1. Distributed aspects of the API

One of the important aspect in distributed model is how the data are exchanged between applications running at different locations. The Pyro4 communication layer allows to exchange data in terms of get and set API methods in two ways. The communication layer automatically takes care of any object that is passed around through remote method calls. The receiving side of a call can receive either a local copy of the remote data or the representation of the remote data (Proxy).

- The communication in terms of exchanging local object copies can be less efficient than communication with remote objects directly, and should be used for objects with low memory footprint. One potential advantage is that the receiving side receives the copy of the data, so any modification of the local copy will not affect the source, remote data. Also multiple method invocation on local objects is much more efficient, compared to costly communication with a remote object.

- On the other hand, the data exchange using proxies (references to remote data) does not involve the overhead of creating the object copies, which could be prohibitively large for complex data structures. Also, when references to the remote objects are passed around, the communication channel must be established between receiving side and remote computer owning the actual object, while passing local objects requires only communication between caller and receiver.

Both approaches have their pros and cons and their relative efficiency depends on actual problem, the size of underlying data structures, frequency of operations on remote data, etc.

Pyro4 will automatically take care of any Pyro4 objects that you pass around through remote method calls. If the autoproxying is set to on (AUTOPROXY = True by default), Pyro4 will replace objects by a proxy automatically, so the receiving side can call methods on it and be sure to talk to the remote object instead of to a local copy. There is no need to create a proxy object manually, a user just has to register the new object with the appropriate daemon. This is a very flexible mechanism, however, it does not allow explicit control on the type of passed objects (local versus remote).

Typically, one wants to have explicit control whether objects are passed as proxies or local copies. The get methods (such as *getProperty*, *getField*) should not register the returned object at the Pyro4 daemon. When used, the remote receiving side obtains the local copy of the object. To obtain the remote proxy, one should use *getFieldURI* API method, which calls *getField* method, registers the object at the server daemon and returns its URI. The receiving side then can obtain a proxy object from URI. This is illustrated in the following code snippet:

```
field_uri = Solver.getFieldURI(FieldID.FID_Temperature, 0.0)
field_proxy = Pyro4.Proxy(uri)
```

8.2. Requirements for distributed computing

To enable the discovery of remote objects a nameserver service is required, allowing to keep track of individual objects in network. It also allows to use readable uniform resource identifiers (URI) instead of the need to always know the exact object id and its location.

The platform is designed to work on virtually any distributed platform, including grid and cloud infrastructure. For the purpose of performing simulations within a project, it is assumed that individual simulations and therefore the individual simulation packages will be distributed over the network, running on dedicated servers provided by individual partners, forming grid-like infrastructure.

According to requirements specified in D1.2 Software Requirements Specification Document for Cloud Computing [2], different functional requirements have been defined, with different levels of priorities. Typical requirements include services for resource allocation, access and license control, etc. In the project, we decided to follow two different strategies, how to fulfill these defined requirements. The first one is based on developing custom solution for resource

allocation combined with access control based on standardized SSH technology based on public key cryptography for both connection and authentication. It uses platform distributed object technology and this allows its full integration in the platform. This solution is intended to satisfy only the minimum requirements, but its setup and operation is easy. Its setup does not require administrative rights and can be set up and run using user credentials. The second approach is based on established condor middleware. This solution provides more finer control over all aspects. On the other hand, its setup is more demanding. The vision is to allow the combination of both approaches. Both approaches and their requirements are described in following sections.

8.3. Internal platform solution - JobManager resource allocation

This solution has been developed from a scratch targeting fulfilment of minimal requirements only while providing simple setup. The resource allocation is controlled by *JobManager*. Each computational server within a platform should run an instance of *JobManager*, which provides services for allocation of application instances based on user request and monitoring services.

The *JobManager* is implemented as python object like any other platform components and is part of platform source code. It is necessary to create an instance of *JobManager* on each application server and register it on the platform nameserver to make it accessible for clients running simulation scenarios. This allows to access *JobManager* services using the same Pyro technology, which makes the resource allocation to be part of the simulation scenario. Typically, the simulation scenario script first establishes connection to the platform nameserver, which is used to query and create proxies of individual *JobManagers*. The individual *JobManagers* are subsequently requested to create the individual application instances (using *allocateJob* service) and locally represented by corresponding proxy objects. Finally, the communication with remote application instances can be established using proxies created in the previous step, see Fig. 10 illustrating typical work flow in the distributed case.

The job manager has only limited capability to control allocated resources. In the present implementation, the server administrator can impose the limit on number of allocated applications. The configuration of the jobmanager requires only simple editing of configuration file. The individual applications are spawned under new process to enable true concurrency of running processes and avoid limitations of Python related to concurrent thread processing.

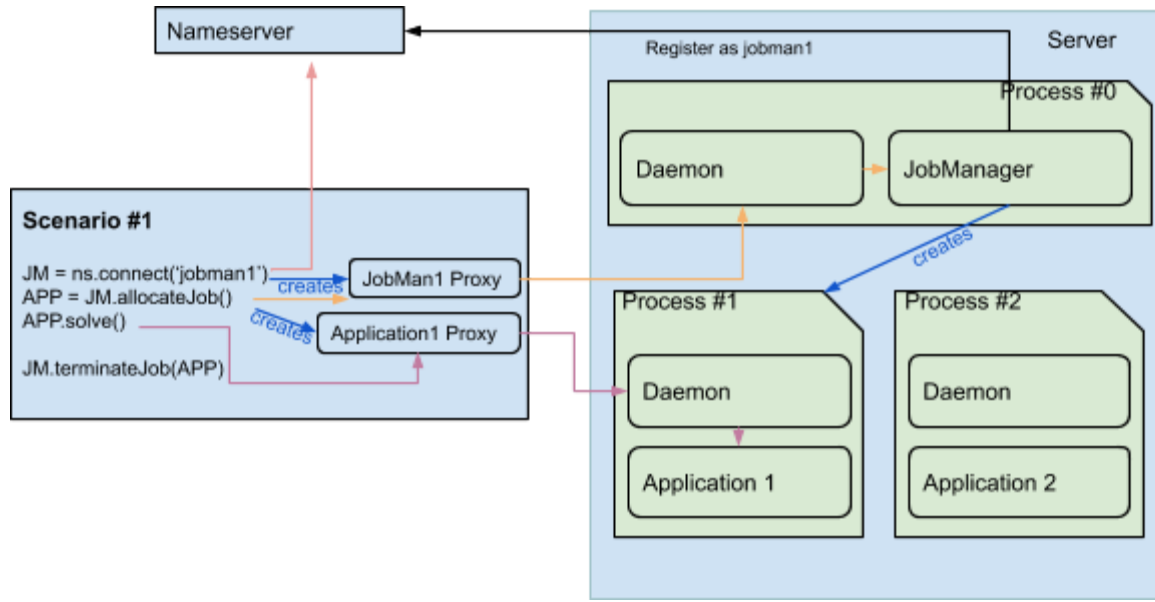


Figure 10: Typical control flow with resource allocation using JobManager.

The status of individual job managers can be monitored with the `jobManStatus.py` script, located in `tools` subdirectory of the platform distribution. This script displays the status of individual jobs currently running, including their run time and user information. The information displayed is continuously refreshed, see Fig. 11.

```
bp@jaja: /home/bp/Documents/projects/MMP/mupif.git/tools
bp@jaja: /home/bp/Documents/projects/MM... x bp@jaja: /home/bp/Documents/projects/MM... x
MuPIF Remote JobMan MONITOR 11:14:50
JobManager:Mupif.JobManager@demo
-----
JobID          Port  user@host      time
5@DemoApplication 9094  bp@jaja        00:00:15
7@DemoApplication 9096  bp@jaja        00:00:02
[q]uit
```

Figure 11: Screenshot of Job Manager monitoring tool

The internal jobManager does not provide any user authentication service at the moment. The user access is assumed to be controlled externally, using ssh authorization. For example, to establish the authorized connection to a remote server and platform services (jobManager) using a ssh tunnel, a valid user credentials for the server are required. The secured, authenticated connection is realized using setting up ssh tunnel establishing a secure and trusted connection to a server. The ssh connections can be authorized by traditional user/passwords or by accepting public ssh keys generated by individual clients and send to server administrators. More details are given in a Section on SSH tunneling.

The status of individual computational servers can be monitored online using the provided monitoring tool. A simple ping test can be executed, verifying the connection to the particular server and/or allocated application instance.

8.3.1. Setting up a Job Manager

The skeleton for application server is distributed with the platform and is located in *examples/Example04-JobMan-distrib*. The following files are provided:

- `server.py`: The implementation of application server. It starts JobManager instance and corresponding daemon. Most likely, no changes are required.
- `serverConfig.py`: configuration file for the server. The individual entries have to be customized for particular server. Follow the comments in the configuration file. In the example, the server is configured to run on Unix-based system.
- `JobMan2cmd.py`: python script that is started in a new process to start the application instance and corresponding daemon. Its behaviour can be customized by `Config.py`.
- `test.py`: Python script to verify the jobManager functionality.
- `clientConfig.py`: configuration file for client code (simulation scenarios). The client can run on both Unix / Windows systems, configuring correctly ssh client.

The setup requires to install the platform, as described in [3. Platform installation](#). Also, the functional application API class is needed. Fig. 12 shows the flowchart with a JobManager using ssh tunnels (VPN is showed further).

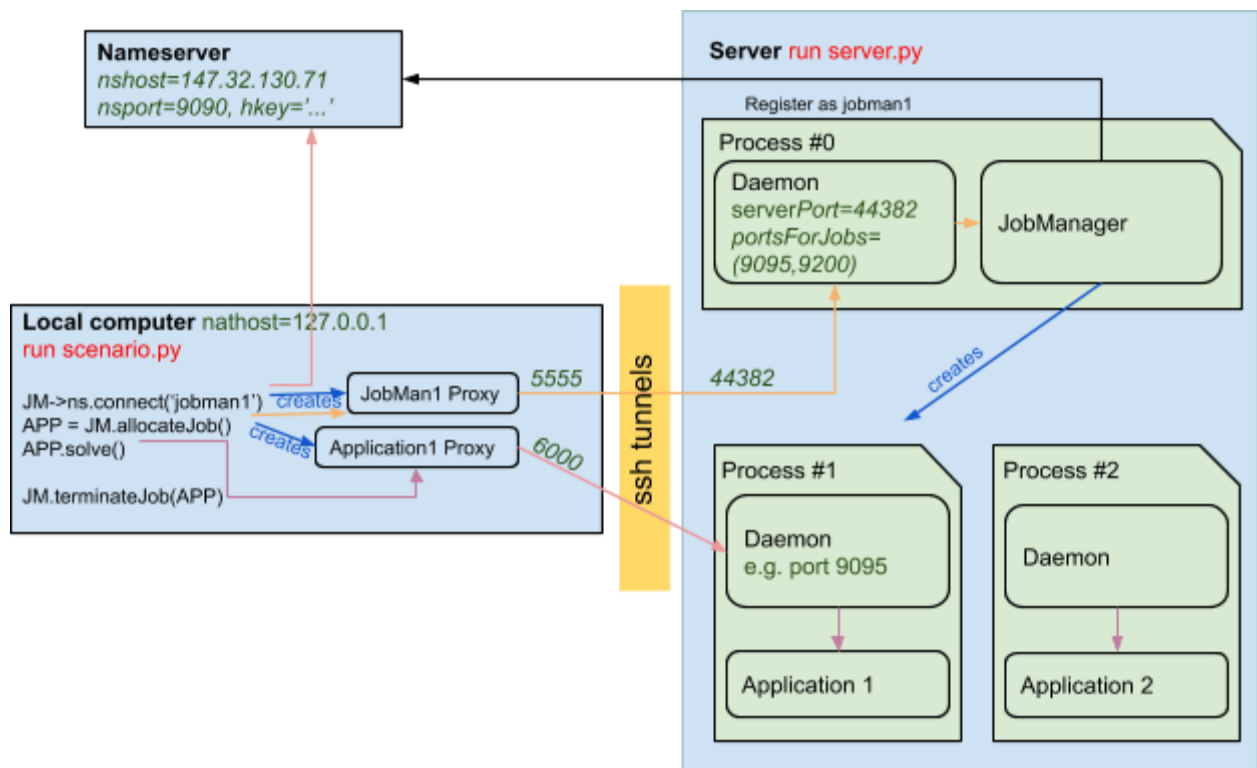


Figure 12: *Example04-JobMan-distrib* displaying ports and tunnels in a distributed setup using ssh tunnels.

The recommended procedure to set up job manager for your server is to create a separate directory, where you will copy the `server.py` and `serverConfig.py` files from *examples/Example04-JobMan-distrib* an directory and customize settings in `serverConfig.py`.

Simpler situation exists for VPN network setup where no ssh tunnels needs to be allocated and all communication runs on a local-like network.

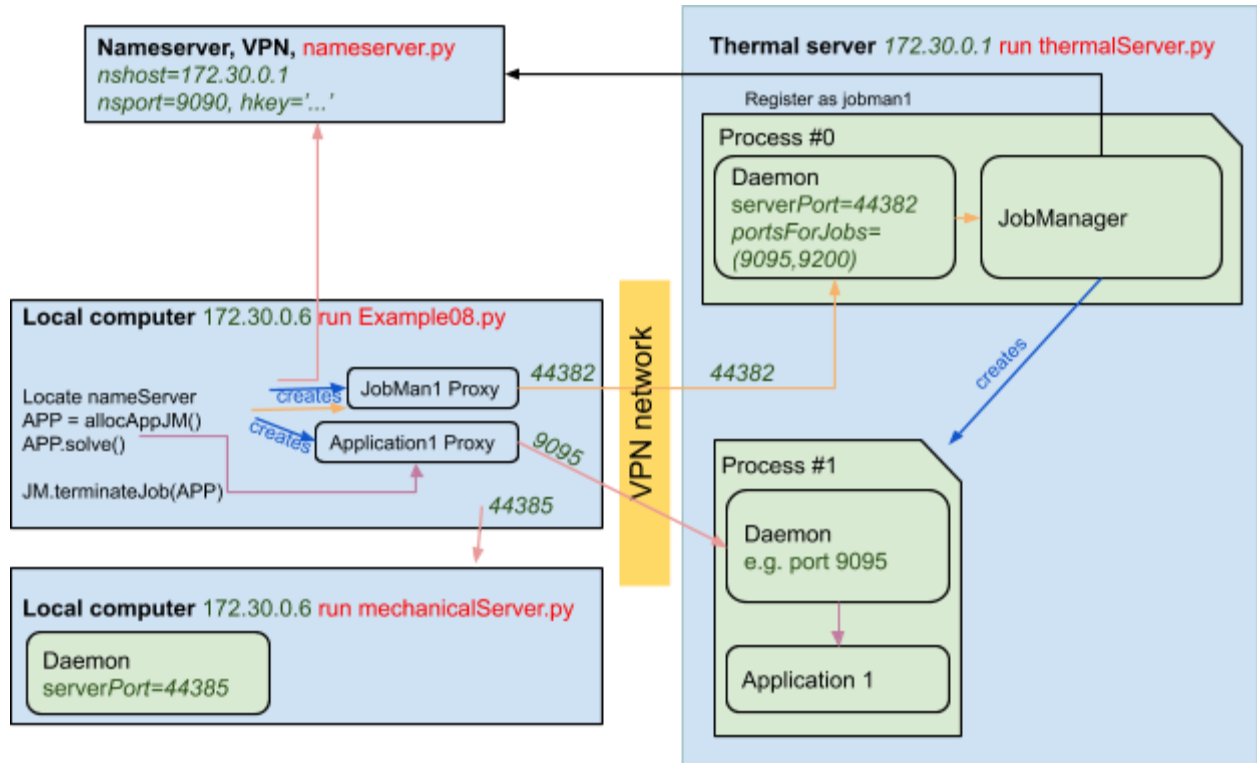


Figure 13: *Example16* thermo-mechanical analysis displaying ports and tunnels in a distributed setup using VPN.

8.3.2. Configuration

The configuration of the job manager consists of editing the configuration file (`thermalServerConfig.py`). The following variables can be used to customize the server settings:

Variable	Description
server	hostname or IP address of the application server, i.e. <code>server='147.32.130.137'</code> . <code>serverPort</code> where the server listens to. Nats needs to be defined in ssh mode only.
serverUserName	user name to establish ssh connection to server, i.e. <code>serverUserName='mmp'</code>
serverPort	Server port where job manager daemon listens, i.e., <code>serverPort=44361</code> .
serverNathost, serverNatport	Port reported by nameserver used to establish tunnel to destination JobManager port (<code>jobManPort</code>), i.e. <code>serverNatpo=5555</code>

jobManName	Name used to register jobManager at nameserver, i.e, jobManName='Mupif.JobManager@micress'
portsForJobs	List of dedicated ports to be assigned to application processes (recommended to provide more ports than maximum number of application instances, as the ports are not released immediately by operating system, see jobManMaxJobs) Example: portsForJobs=(9091, 9092, 9093, 9094)
maxJobs	Maximum number of jobs that can be running at the same time, e.g. maxJobs = 20
jobManWorkDir	Path to JobManager working directory. In this directory, the subdirectories for individual jobs will be created and these will become working directories for individual applications. Users can upload/download files into these job working directories. Note: the user running job manager should have corresponding I/O (read/write/create) permissions.
applicationClass	Class name of the application API class. The instance of this class will be created when new application instance is allocated by job manager. The corresponding python file with application API definition need to be imported.
applicationInitialFile	Initial file read by an application.

The individual ports can be selected by the server administrator, the ports from range 1024-49152 can be used by users / see IANA (Internet Assigned Numbers Authority).

To start an application server run (*Example04-JobMan-distrib*):

```
$ python3 server.py
```

The command logs on screen and also in the server.log logfile the individual requests.

The status of the application server can be monitored on-line from any computer (provided you have established ssh connection to server) using tools/jobManStatus.py monitor. To start monitoring, run e.g. the following command:

```
$ python3 jobManStatus.py -j Mupif.JobManager@Example -n 127.0.0.1
```

The -j option specifies the jobmanager name (as registered in pyro nameserver), -h determines the hostname where jobmanager runs, -p determines the port where jobmanager is listening, -n is hostname of the nameserver, -r is the nameserver port, -k allows to set PYRO hkey, -t enforces the ssh tunnelling, and -u determines the username to use to establish ssh connection on the server, see Fig. 14.

```
File Edit View Search Terminal Help
MuPIF Remote JobMan MONITOR 20:31:38
Mupif.JobManager@Example on 127.0.0.1
-----
JobID          Port  user@host          time
1@Mupif.JobManager@Example  9000  smilauer@smilauer-ThinkPad-L500:01:26
[q]uit
```

Figure 14: Testing job manager in a simple setup

There is also a simple test script (`tools/jobManTest.py`), that can be used to verify that the installation procedure was successful. It contact the application server and asks for new application instance.

8.4. Securing the communication using SSH tunnels

8.5. Setting up ssh server

SSH server provides functionalities which generally allows to

- Securely transfer encrypted data / streams
- Securely transfer encrypted files (SFTP)
- Set up port forwarding via open ports, so called tunneling, allowing to get access to dedicated ports through a firewall in between
- Remote command execution
- Forwarding or tunneling a port
- Securely mounting a directory on a remote server (SSHFS)

Ssh server is the most common on Unix systems, *freeSSHd* server can be used on Windows free of charge. The server usually requires root privileges for running. Ssh TCP/UDP protocol uses port 22 and uses encrypted communication by default.

Connection to a ssh server can be carried out by two ways. A user can authenticate by typing username and password. However, MuPIF prefers authentication using asymmetric private-public key pairs since the connection can be established without user's interaction and password typing every time. Fig. 15 shows both cases.

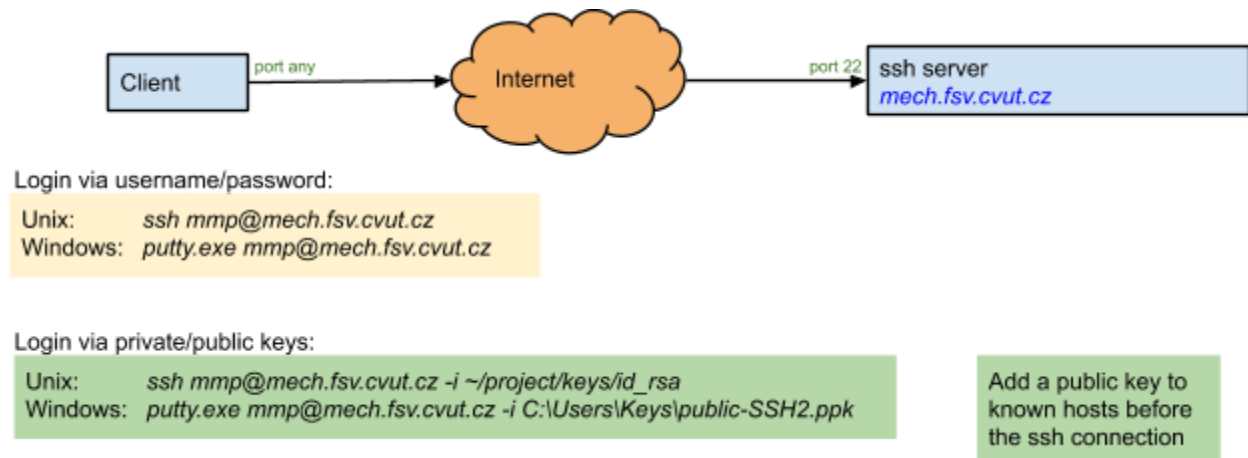


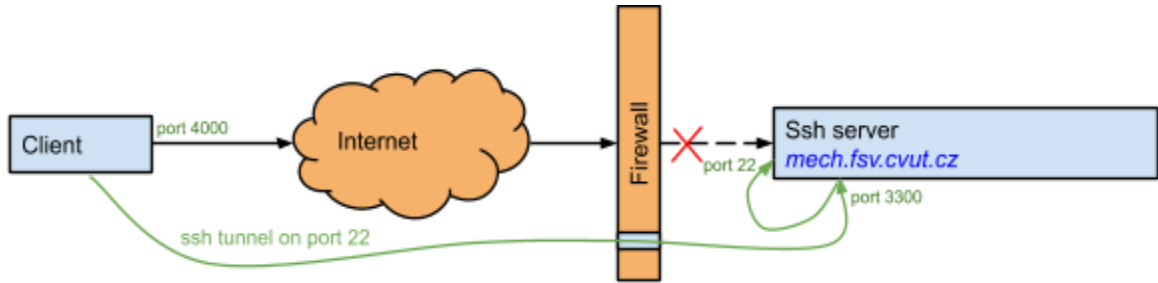
Figure 15: Connection to a ssh server using username/password and private/public keys

Private and public keys can be generated using commands `ssh-keygen` for Unix and `puttygen.exe` for Windows. Ssh2-RSA is the preferred key type, no password should be set up since it would require user interaction. Keys should be stored in ssh2 format (they can be converted from existing openSSH format using `ssh-keygen` or `puttygen.exe`). Two files are created for private and public keys; Unix `id_rsa` and `id_rsa.pub` files and Windows `id_rsa.ppk` and `id_rsa` files. Private key is a secret key which remains on a client only.

Authentication with the keys requires appending a public key to the ssh server. On Unix ssh server, the public key is appended to e.g. `mech.fsv.cvut.cz:/home/user/.ssh/authorized_keys`. The user from a Unix machine can log in without any password using a ssh client through the command

```
ssh user@mech.fsv.cvut.cz -i ~/project/keys/id_rsa
```

Ssh protocol allow setting up port forwarding via port 22, so called tunneling. Such scenario is sketched in Fig. 16, getting through a firewall in between. Since the communication in distributed computers uses always some computer ports, data can be easily and securely transmitted over the tunnel.



Unix: `ssh -L 4000:mech.fsv.cvut.cz:3300 mmp@mech.fsv.cvut.cz`
 Windows: `putty.exe -L 4000:mech.fsv.cvut.cz:3300 mmp@mech.fsv.cvut.cz`

Figure 16: Creating a ssh forward tunnel

8.6. Example of distributed scenario with ssh tunneling

The process of allocating a new instance of remote application is illustrated on adapted version of the local thermo-mechanical scenario, already presented in [7. Developing user workflows](#). First, the configuration file is created containing all the relevant connection information, see Listing 3.

```

#Network setup configuration
import sys, os, os.path
import Pyro4
# Pyro config
Pyro4.config.SERIALIZER="pickle"
Pyro4.config.PICKLE_PROTOCOL_VERSION=2 #to work with python 2.x and 3.x
Pyro4.config.SERIALIZERS_ACCEPTED={'pickle'}
Pyro4.config.SERVERTYPE="multiplex"

#Absolute path to mupif directory - used in JobMan2cmd
mupif_dir = os.path.abspath(os.path.join(os.getcwd(), "../.."))
sys.path.append(mupif_dir)

import logging

#NAME SERVER
nshost = '147.32.130.71' #IP/name of a name server
nsport = 9090 #Port of name server
hkey = 'mmp-secret-key' #Password for accessing nameServer and applications

#Remote server settings
server = '147.32.130.71' #IP/name of a server's daemon
serverPort = 44382 #Port of server's daemon
serverNatHost = '127.0.0.1' #Nat IP/name (necessary for ssh tunnel)
serverNatPort = 5555 #Nat port (necessary for ssh tunnel)

jobManName='Mupif.JobManager@Example' #Name of job manager
appName = 'MuPIFServer' #Name of application

#JobManager setup
portsForJobs=( 9095, 9200 ) #Range of ports to be assigned on the server to jobs
jobNatPorts = list(range(6000, 6050)) #NAT client ports used to establish ssh cons
maxJobs=4 #Maximum number of jobs
#Auxiliary port used to communicate with application daemons on a local computer

```



```

socketApps=10000
jobManWorkDir='.' #Main directory for transmitting files

jobMan2CmdPath = "../tools/JobMan2cmd.py" #Path to JobMan2cmd.py

#CLIENT
serverUserName = os.getenv('USER')

#ssh client params to establish ssh tunnels
if(sys.platform.lower().startswith('win')):#Windows ssh client
    sshClient = 'C:\\Program Files\\Putty\\putty.exe'
    options = '-i L:\\.ssh\\mech\\id_rsa.ppk'
    sshHost = ''
else:#Unix ssh client
    sshClient = 'ssh'
    options = '-oStrictHostKeyChecking=no'
    sshHost = ''

```

Listing 3: Simple example illustrating simulation scenario

Remote connection by ssh is done by setting *-m 1* after the script which picks up correct configuration. It is explained on *Example08-transiTM-JobMan-distrib*. First, the simulation scenario connects to the nameserver and subsequently the handle to thermal solver allocated by the corresponding job manager is created using *PyroUtil.allocateApplicationWithJobManager* service. This service first obtains the remote handle of the job manager for thermal application, requests allocation of a new instance of thermal solver, returning an instance of RemoteModel decorator, a class which encapsulate all the connection details (opened connections, established ssh tunnels, etc.) and acts as proxy to the allocated remote application instance.

8.7. Advanced SSH setting

When a secure communication over ssh is used, then typically a steering computer (a computer executing top level simulation script/workflow) creates connections to individual application servers. However, when objects are passed as proxies, there is no direct communication link established between individual servers. **This is quite common situation, as it is primarily the steering computer and its user, who has necessary ssh-keys or credentials to establish the ssh tunnels from its side, but typically is not allowed to establish a direct ssh link between application servers.** The solution is to establish such a communication channel transparently via a steering computer, using forward and reverse ssh tunnels. The platform provides handy methods to establish needed communication patterns (see *PyroUtil.connectApplications* method and refer to *Example07-stacTM-JobMan-distrib* for an example).

As an example, consider the simulation scenario composed of two applications running on two remote computers as depicted in Fig. 17. The Pyro4 daemon on server 1 listens on communication port 3300, but the nameserver reports the remote objects registered there as listening on local ports 5555 (so called NAT port). This mapping is established by ssh tunnel

between client and the server1. Now consider a case, when application2 receives a proxy of object located on server1. To operate on that object the communication between server 1 and server 2 needs to be established, again mapping the local port 5555 to target port 3300 on server1. Assuming that steering computer already has an established communication link from itself to Application1 (realized by ssh tunnel from local NAT port 5555 to target port 3300 on the server1), an additional communication channel from server2 to steering computer has to be established (by ssh tunnel connecting ports 5555 on both sides). In this way, the application2 can directly work with remote objects at server 1 (listening on true port 3300) using proxies with NAT port 5555.

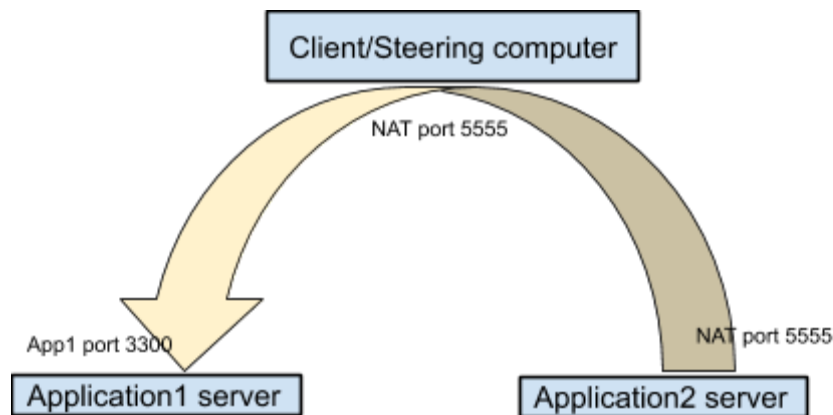


Figure 17: Establishing a communication link between two application servers via SSH tunnels.

8.4.4. Troubleshooting SSH setup

- Verify that the connection to nameserver host works:
 - `ping name_server_hostname`
- Run the `jobManTest.py` with additional option “-d” to turn on debugging output, examine the output (logged also in `mupif.log` file)
- Examine the output of server messages printed on screen and/or in file `server.log`

8.8. Using Virtual Private Network (VPN)

8.8.1. Generalities

This section only provides background for VPN and can be skipped. The standard way of node communication in MuPIF is to use SSH tunnels. SSH tunnels have the following advantages:

- No need for administrator privileges.
- Often the way for remotely accessing computers which are already in use.
- Easy traversal of network firewalls (as long as the standard port 22 is open/tunneled to the destination).

They also have some disadvantages:

- Non-persistence: the tunnel has to be set up every time again; if connection is interrupted, explicit reconnection is needed, unless automatic restart happens, e.g. [autossh](#).

The tunnel is only bi-directional and does no routing; thus if A-B is connected and B-C is connected, it does not imply C is reachable from A. Though, it is possible to create a multi-hop tunnel by chaining `ssh` commands.

VPN is an alternative to SSH tunnels, providing the encryption and authorization services. The VPNs work on a lower level of communication (OSI Layer 2/3) by establishing “virtual” (existing on the top of other networks) network, where all nodes have the illusion of direct communication with other nodes through TCP or UDP, which have IP addresses assigned in the virtual network space, see Fig. 18. The VPN itself communicates through existing underlying networks, but this aspect is not visible to the nodes; it includes data encryption, compression, routing, but also authentication of clients which may connect to the VPN. [OpenVPN](#) is a major implementation of VPN, and is supported on many platforms, including Linux, Windows, Android and others.

Using VPN with MuPIF is a trade-off where the infrastructure (certificates, VPN server, ...) is more difficult to set up, but clients can communicate in a secure manner without any additional provisions - it is thus safe to pass unencrypted data over the VPN, as authentication has been done already; in particular, there is no need for SSH tunnels inside MuPIF.

Note that all traffic exchanged between VPN clients will go through the OpenVPN server instance; the connection of this computer should be fast enough to accommodate all communication between clients combined.

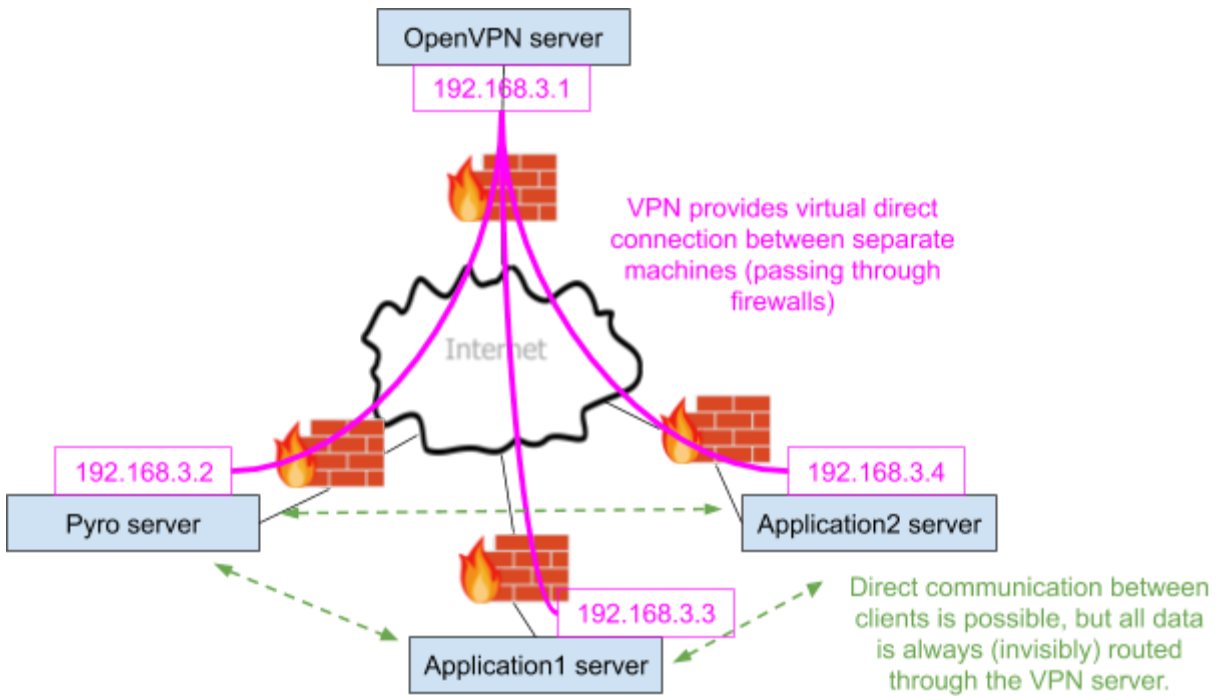


Figure 18: VPN architecture

8.8.2. Setup

Setting up the VPN is generally more difficult than ssh tunnels. It comprises the following:

- Communication ports reachable by all clients must be set up as a part of the infrastructure (usually on a static & public IP address); this involves opening ports in firewalls, and most network administrators are not very keen to do that. While these are configurable, the default is UDP 1194 for client access; often TCP 443 is also (ab)used (it is commonly and by standard used for HTTPS).
- Running the OpenVPN daemon on the server; server configuration is not overly complicated, there are in fact many good tutorials available.
- Distributing OpenVPN configuration files (usually ending .ovpn) to the clients.
- Clients have to connect to the VPN whenever they want to communicate with the network - this can be done from the command-line or using graphical interfaces.

Whenever a client connects to the OpenVPN server, the following happens:

1. The client is authenticated, either via username/password or certificate.
2. The client is handed an IP address from the VPN range, as specified by ifconfig-pool configuration option, or assigned a fixed IP based on the client configuration (client-config-dir), see [OpenVPN Addressing](#).
3. The client's OS assigns the IP address to a virtual network adapter (tun0, tun1 etc in Linux) and sets IP routing accordingly. Depending on server configuration, all non-local traffic (such as to public internet hosts) may be routed through the VPN, or only traffic for VPN will go through the VPN. At this moment, other clients of the VPN become visible to

the new client, and vice versa (it is client's responsibility to firewall the VPN interface, if desired).

There are example scripts to generate OpenVPN configuration for MuPIF in *tools/vpn*. The script generates certificate authority and keys used for authentication of server and clients, and also for traffic encryption; those files must be slightly hand-adjusted for real use afterwards. The recommended configuration for MuPIF is the following (non-exhaustive; the tutorial from digitalocean (www.digitalocean.com/.....) explains most of the procedure).

1. Use the usual "subnet" network topology.
2. IP addresses within the VPN may be assigned from the address pool, but at least some machines should have fixed IP - this can be done using the `client-config-dir` option. In particular, the Pyro nameserver should have a well-known and stable IP address so that the client configuration does not have to change; the best is to run the OpenVPN server on the same computer where Pyro runs, then the IP address will be stable.
3. Only in-VPN traffic should be routed through the VPN (thus the `redirect-gateway` option should not be used); communication of clients with Internet will go through the usual ISP route of each client.
4. Firewall facing internet should allow UDP traffic on port 1194. Optionally, other port can be used (even non-OpenVPN port, like TCP/443, which is normally used for HTTPS). All traffic on the `tun0` (or other number) interfaces should be allowed; one can use the `-i tun+` option of iptables to apply a rule to any interface of which name starts with `tun`.
5. `Keepalive` option can be used to increase network reliability (functions as both heart-beat & keep-alive).
6. Authentication can be done using username & password, but key-based authentication (client keys must be distributed to clients) is recommended.
7. The server is started either as a daemon (through `init.d` or `systemd`) or from the commandline, in which case "Initialization Sequence Completed" will be shown when ready to serve clients.

Client configuration:

1. If the configuration is distributed as `.ovpn` file with embedded keys, the VPN can be activated from command-line by issuing `sudo openvpn --config client.ovpn`. The client will say Initialization Sequence Completed after successful connection to the VPN. Use `Ctrl-C` to terminate the client and disconnect from the VPN.
2. The GUI of NetworkManager can import the configuration and use it, but not in all cases (embedded keys seem to be the problem), in which case the `.ovpn` file can only contain filenames where the keys/certs are stored, or the configuration can be created by hand through the NetworkManager GUI.
3. Connection to the VPN can be verified by issuing `"ip addr show"` which should show the `tun0` (or similar) interface with an IP assigned from the OpenVPN server pool.

8.8.3. Example of simulation scenario using VPN

The process of allocating a new instance of remote application is illustrated on adapted version of the local thermo-mechanical scenario, already presented in [7. Developing user workflows](#). VPN mode can be enforced by issuing commands with `-m 2` at the end. Refer to *examples/Example07-stacTM-JobMan-distrib*.

8.8.4. Online Monitoring tool

To monitor the status of VPN network as well as status of the MuPIF infrastructure, an online monitoring tool has been developed. It is based on OpenVPN-monitor tool, which monitors the status of VPN server and connected VPN clients. It has been extended to display stats about status of MuPIF infrastructure. It shows the status of the nameserver, the list of registered jobManagers, their connection information and number of running tasks. The monitoring tool is accessible from any web browser running on a computer connected to the VPN network.

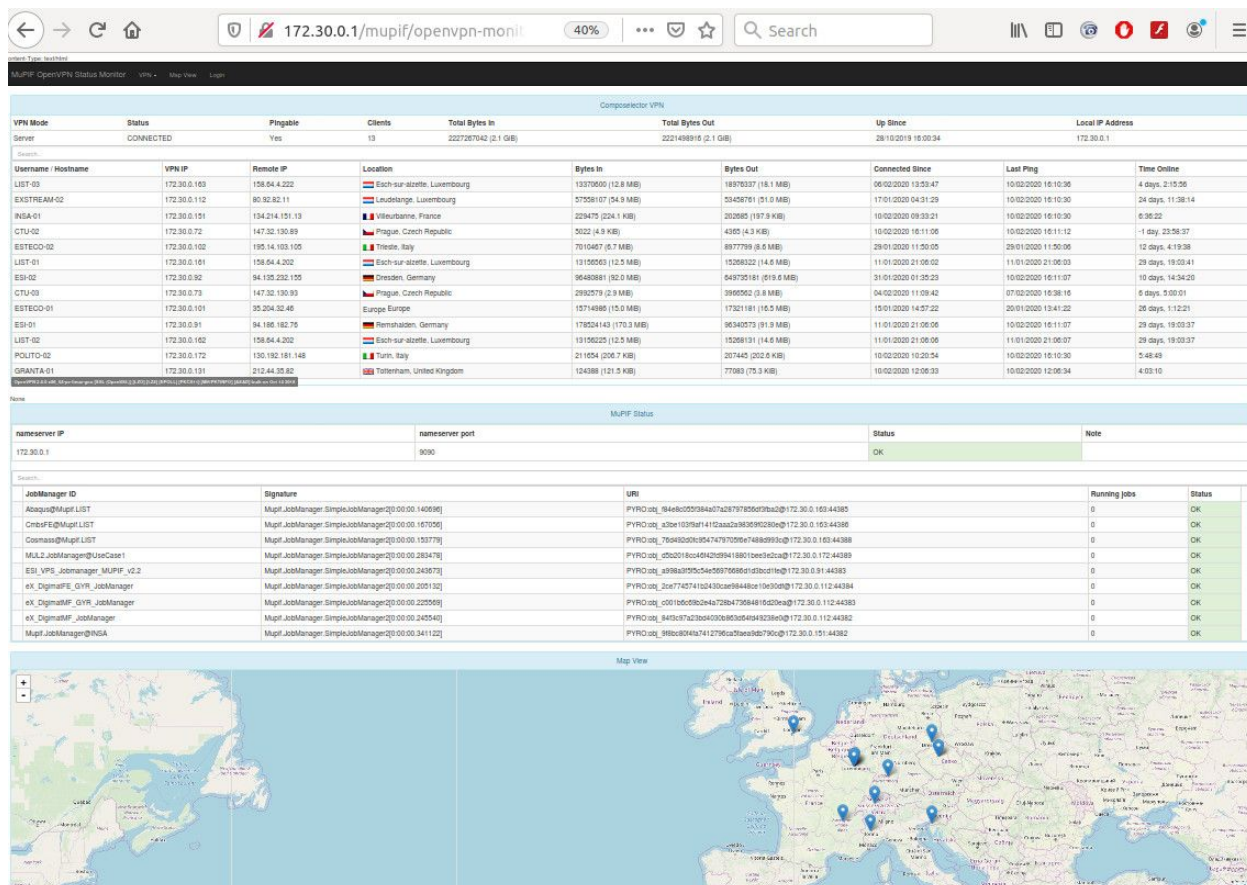


Figure 19: Screenshot of VPN and platform monitoring tool

9. Acknowledgements

The development of MuPIF has been funded by Grant Agency of the Czech Republic - Projects No. P105/10/1402.

The development of the platform has been funded by FP7 under NMP-2013-1.4-1 call 1.4-1 "Development of an integrated multi-scale modelling environment for nanomaterials and systems by design" with Grant agreement no: 604279, entitled [Multiscale Modelling Platform: Smart design of nano-enabled products in green technologies](#).

The authors would like to acknowledge the EU support from Horizon 2020 Project [Composelector](#), Contract N°: 721105

10. ChangeLog

10.1. Version 1.1 (05/2017)

- Expanded section on workflow implementation, added subsections on workflow templates and workflow as a class. Already describes some concept to be introduced in ver. 2.0 (transparent ssh tunnel handling using decorator classes).
- Added acknowledgement to EU Composelector project.

10.2. Version 2.0.0 (12/2017)

- Updated API doc to MuPIF ver. 2.0
- Metadata support by introducing abstract, top-level *MuPIFObject* class, which is a parent class to all MuPIF components and provides methods to attach and retrieve metadata
- Workflows can be represented as a class derived from *Workflow* class. Workflows share the same API as models (*Model* class). This allows to combine models and workflows and create a complex, hierarchical workflows. Several examples converted to demonstrate workflow-as-a-class concept.
- TimeStep now requires unit information on all attributes, introduced new attribute allowing to set target simulation time.
- Base *Property* class generalized to allow for properties depending on parameters (time, other variables), Properties now strictly unit aware.
- *Field* class now strictly unit aware.
- Added section on VPN online monitoring tool

10.3. Version 2.2.0 (04/2019)

- Updated to MuPIF ver 2.2
- Updated to Python ≥ 3.2
- Section 5.1 extended to cover metadata and metadata schemata
- Updated example numbering
- Updated Pyro4 to version 4.75

10.4. Version 2.3.0 (02/2020)

- Updated to MuPIF ver 2.3

- Updated to Python ≥ 3.5
- Added Particle and ParticleSet classes and corresponding unit tests
- Fixing metadata schemas for Model and Workflow classes
- Email notification of failed JobManager
- Fixing Mesh class for vertex support
- Pilot tests for EMMO (European Materials & Modelling Ontology) under examples/Example06-stacTM-local/emmo*.py

11. References

1. D1.1 Application Interface Specification, MMP Project, 2014.
2. D1.2 Software Requirements Specification Document for Cloud Computing, MMP Project, 2015.
3. Python Software Foundation. Python Language Reference, version 3.5. Available at <http://www.python.org>
4. Pyro - Python Remote Objects, <http://pythonhosted.org/Pyro>
5. B. Patzák, D. Rypl, and J. Kruis. MuPIF – a distributed multi-physics integration tool. Advances in Engineering Software, 60–61(0):89 – 97, 2013 (<http://www.sciencedirect.com/science/article/pii/S0965997812001329>).
6. B. Patzak, V. Smilauer, and G. Pacquaut, accepted presentation & paper “*Design of a Multiscale Modelling Platform*” at the conference Green Challenges in Automotive, Railways, Aeronautics and Maritime Engineering, 25th - 27th of May 2015, Jyväskylä, Finland.
7. B. Patzak, V. Smilauer, and G. Pacquaut, presentation & paper “*Design of a Multiscale Modelling Platform*” at the 15th International Conference on Civil, Structural, and Environmental Engineering Computing, 1st - 4th of September 2015, Prague, Czech Republic.
8. B. Patzak, V. Smilauer: MuPIF reference manual 1.0.0, 2016. Available at www.mupif.org
9. Directorate-General for Research and Innovation (European Commission), [What makes a material function? Let me compute the ways : modelling in H2020 LEIT-NMBP programme materials and nanotechnology projects - Study](#), ISBN: 978-92-79-63185-6 DOI: 10.2777/417118, 2017.
10. The European Materials Modelling Council, <https://emmc.info/>, 2017.
11. JSON Schema, <https://json-schema.org/>, 2019.