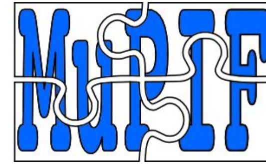



# MuPIF.org Platform User Manual

Authors: B. Patzák, V. Šmilauer[1]

Version 05/2016

## Table of Content


- Table of Content
- Introduction
- Platform installation
    - Prerequisites
        - Windows platforms
        - Linux / Unix (*nix) platforms
        - General requirements
        - Other recommended packages/softwares
    - Installing the MuPIF platform
    - Verifying platform installation
- Platform operations
- Platform APIs
    - Application class
    - Property class
    - Field class
    - Function class
    - TimeStep class
    - Mesh class
    - Cell class
    - Vertex class
    - BoundingBox
    - APIError
- Developing Application Program Interface (API)
- Distributed Model
    - Distributed aspects of the API


[1] Czech Technical University, Faculty of Civil Engineering, Department of Mechanics, Thákurova 7, 166 29, Prague, Czech Republic.

- Requirements for distributed computing
- Internal platform solution - JobManager resource allocation
  - Installation
    - Setting up ssh server
    - Setting up Job Manager
      - Configuration
    - Troubleshooting
  - Simple distributed example using jobManager resource allocation
- Acknowledgements
- References


# 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 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. This essentially allows to manipulate all simulation tools using the same interface. Moreover, as the simulation data are represented by objects as well, the platform is independent on particular data format(s), as the exchanged data (such as fields and properties) can be manipulated using the same abstract interface. Therefore, the focus on services is provided by objects (object interfaces) and not on underlying data itself.

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 recasted into a class implementing Application 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.

# Platform installation

## Prerequisites

### Windows platforms

- We suggest to install Anaconda scientific python package (tested version 2.1): https://store.continuum.io/cshop/anaconda/
- ssh client: putty.exe is recommended, http://www.putty.org/
- optionally ssh key generator: puttygen.exe is recommended, http://www.putty.org/
- optionally ssh server if you need to accept SSH incoming connections and allowing others to be on your system. FreeSSHd server is recommended, http://www.freesshd.com/

### Linux / Unix (*nix) platforms

- The Python (Python 2.x) installation is required. Some functionality depend on vtk python module, that is available in Python 2.x version only.
- You can dowload the python installation package from https://www.python.org/downloads/. Just pick up the latest version in the 2.x series (tested version 2.7.8).
- We recommend to install pip - a tool for installing and managing Python packages. If not already installed as a part of your python distribution, the installation instructions can be found here.
- ssh client (normally included in standard distributions)
- optionally ssh server (required for application server installation)

### General requirements

- MuPIF platform depends/requires Pyro4 (tested version 4.39) and numpy (tested 1.6.2) modules. To install these modules using pip:

```
pip install Pyro4
```

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

```
pip install pyvtk
```

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

```
pip install enum34
```

### Other recommended packages/softwares

- 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/

# Installing the MuPIF platform

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

```
pip 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 git://git.code.sf.net/p/mupif/code mupif-code
```

# Verifying platform installation

The platform installation comes with many examples, that can be used to verify the successful installation. The examples are located in examples subfolder. For example, to run Example01:

```
cd examples/Example01
python Example01.py
```

## 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 [Simple distributed example using JobManager](#).

The presented example in Table 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

# 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();
```

Table 1: Simple example illustrating simulation scenario

The full listing of this example can be found in examples/Example01. The output is illustrated in Figure 1.

```
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$
```

Fig. 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.

# 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. 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 on 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).

## Application class

This abstract class represents an external application and defines its interface. The interface is defined in terms of abstract services for data exchange and steering. Derived classes represent individual simulation tools. The data exchange services consist of methods for getting and registering external properties, fields, and functions, which are represented using corresponding, newly introduce classes. Steering services allow invoking (execute) solution for a specific solution step, update solution state, terminate the application, etc.

| **Service** | **Description** |
|---|---|
| `__init__ (self, file)` | Constructor. Initializes the application.<br>**ARGS:**<br>- file (str): path to application initialization file. |

| | |
|---|---|
| getField(self,fieldID, time) | Returns the requested field at given time. Field is identified by fieldID.<br>**ARGS**:<br>- fieldID (FieldID): identifier<br>- Time (double): target time<br>**Returns**:<br>Returns requested field (Field). |
| setField(self, field) | Registers the given (remote) field in application.<br>**ARGS**:<br>- field (Field): remote field to be registered by the application<br>**Returns**:<br>None |
| getProperty(self,propID, time, objectID=0) | Returns property identified by its ID evaluated at given time.<br>**ARGS**:<br>- propID (PropertyID): property ID<br>- time (double): time when property to be evaluated<br>- objectID (int): identifies object/submesh on which property is evaluated (optional)<br>**Returns**:<br>Returns representation of requested property (Property). |
| setProperty(self, property, objectID=0) | Register given property in the application<br>**ARGS**:<br>- property (Property): the property class<br>- objectID (int): identifies object/submesh on which property is evaluated (optional)<br>**Returns**:<br>None |
| getFunction(self,funcID, objectID=0) | Returns function identified by its ID<br>**ARGS**:<br>- funcID (FunctionID): function ID<br>- objectID (int): identifies optional object/submesh |

| | |
|---|---|
| | **Returns**:<br>Returns requested function(Function) |
| setFunction(self,func, objectID=0) | Register given function in the application<br>**ARGS**:<br>- func(Function): function to register<br>- objectID (int): identifies optional object/submesh |
| getMesh (self, tstep) | Returns the computational mesh for given solution step.<br>**ARGS**:<br>- tstep(TimeStep): solution step<br>**Returns**:<br>Returns the representation of mesh (Mesh) |
| solveStep(self, tstep, stageID=0, runInBackground=False) | Solves the problem for a given time step. Evaluates the solution from actual state to given time.<br>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, finishStep is called and then the actual state has to be updated.<br>Solution can be split into individual stages identified by optional stageID parameter. In between the stages, the additional data exchange can be performed. See also wait and isSolved services.<br><br>**ARGS**:<br>- tstep(TimeStep): solution step<br>- stageID(int): optional argument identifying solution stage<br>- runInBackground(bool): if set to True, the solution will run in background (in separate thread), if supported.<br>**Returns**: None |
| wait(self) | Wait until solve is completed when executed in background.<br><br>**Returns**: None |
| isSolved(self) | Returns true or false depending whether solve has completed when executed in background. |

| | |
|---|---|
| | **Returns**: (Boolean) |
| finishStep(self, tstep) | Called after a global convergence within a time step.<br>**ARGS**:<br>- tstep(TimeStep): solution step<br><br>**Returns**: None |
| getCriticalTimeStep(self) | Returns the actual (related to the current state) critical time step increment (double).<br>**Returns**:<br>Critical time step (double) |
| getAssemblyTime(self, tstep) | Returns the assembly time related to a given time step. The registered fields (inputs) should be evaluated in this time.<br>**ARGS**:<br>- tstep (TimeStep): solution step<br>**Returns**:<br>Assembly time (double) |
| storeState(self, tstep) | Store the solution state of an application.<br>**ARGS**:<br>- tstep(TimeStep): solution step<br><br>**Returns**: None |
| restoreState(self, tstep) | Restore the saved state of an application.<br>**ARGS**:<br>- tstep(TimeStep): solution step<br><br>**Returns**: None |
| terminate(self) | Terminates the application.<br><br>**Returns**: None |
| getAPIVersion(self) | Returns the supported API version.<br>**Returns**:<br>API version (int) |

## 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** |
|---|---|
| `__init__(self, value, propID, valueType, time, objectID=0)` | Constructor, initializes the property.<br>**ARGS:**<br>- 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.<br>- propId (PropertyID): property ID<br>- valueType (ValueType): type of property value<br>- time (double): time<br>- objectID (int): optional ID of problem object / subdomain to which property is related. |
| getValue(self) | Returns the value of property in a tuple.<br>**Returns:**<br>Property value as array (tuple) |
| getPropertID(self) | Returns type of property.<br>**Returns:**<br>Receiver property ID (PropertyID) |
| getObjectID(self) | Returns property objectID.<br>**Returns:**<br>ID of related object (int) |

# 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** |
| --- | --- |
| `__init__(self, mesh, fieldID, valueType, time, values=None)` | Constructor. Initializes the field instance.<br>**ARGS**:<br>- mesh (Mesh): Instance of Mesh class representing underlying discretization.<br>- fieldID (FieldID): field type<br>- valueType (ValueType): type of field values<br>- time (double): time<br>- values (tuple): field values, usually at mesh vertices (format dependent of particular field type) |
| getMesh(self) | Returns representation of underlying discretization.<br>**Returns**:<br>Reference to associated mesh (Mesh) |
| getValueType(self) | Returns type of field values (ValueType) of the receiver.<br>**Returns**:<br>(ValueType) |
| getFieldID(self) | **Returns**:<br>Field ID (FieldID) |
| evaluate(self, position, eps=0.001) | Evaluates the receiver at given spatial position.<br>**ARGS**:<br>- position (tuple, list of tuples): 3D position vector or list of position vectors<br>- eps(double): Optional tolerance<br>**Returns**:<br>Receiver value or list of values evaluated at given position(s) (tuple, list of tuples). |

| | |
|---|---|
| getValue(self, componentID) | Returns the value associated to given component (vertex or cell IP, implementation dependent).<br>**ARGS**:<br>- componentID (tuple): identifies the component (vertexID) or (CellID, IPID)<br>**Returns**:<br>component value (tuple) |
| setValue(self, componentID, value) | Sets the value associated to given component (vertex or cell IP). Note, that the field values are updated after a commit method is invoked.<br>**ARGS**:<br>- componentID (tuple): The componentID is a tuple: (vertexID) or (CellID, IPID)<br>- value(tuple): Component value<br>**Returns**:<br>None |
| commit(self) | Commits the recorded changes (via setValue method).<br>**Returns**: None |
| merge(self, field) | 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.<br>**ARGS**:<br>- field (Field): field to merge<br>**Returns**: None |
| field2VTKData (self) | Returns VTK representation of the receiver.<br>**Returns**:<br>VTK dataset (VTKDataSource) |

## 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 |
| --- | --- |
| `__init__(self,funcID, objectID=0)` | Constructor. Initializes the function.<br>**ARGS**:<br>- funcID (FunctionID): function ID<br>- objectID (int): optional ID of associated subdomain. |
| `evaluate (self, d)` | 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. Dictionaries consist of pairs (called items) of keys and their corresponding values.<br><br>Example:<br>d={'x':(1,2,3), 't':0.005} initializes dictionary containing tuple (vector) under 'x' key, double value 0.005 under 't' key.<br>Some common keys:<br>- 'x': position vector<br>- 't': time<br>**ARGS**:<br>- d (dictionary): dictionary containing function arguments (number and type depends on particular function)<br>**RETURNS**:<br>function value (tuple) evaluated for given parameters |
| `getID (self)` | Returns reciver's ID.<br>**Returns**:<br>id (FunctionID) |
| `getObjectID(self)` | **Returns**:<br>returns receiver's object ID (int) |

# TimeStep class

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

| Service | Description |
|---|---|
| `__init__(self, t, dt, n=1)` | Constructor. Initializes the new time step.<br>**ARGS:**<br>- t (double): time<br>- dt (double): step length (time increment)<br>- n (int): time step number |
| `getTime(self)` | **Returns:**<br>Time step time (double) |
| `getTimeIncrement(self)` | **Returns:**<br>time increment (double) |
| `getNumber(self)` | **Returns:**<br>receiver's number (int) |

## 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 |
|---|---|
| `__init__(self)` | Constructor, creates an empty mesh. |

| | |
|---|---|
| copy(self) | 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.<br>**Returns**:<br><br>Copy of receiver (Mesh) |
| getNumberOfVertices(self) | **Returns**:<br>Number of Vertices (int) |
| getNumberOfCells(self) | **Returns**:<br>Number of Cells |
| getVertex(self, i) | Returns i-th vertex (i corresponds to a vertex number, not a label).<br>**Returns**:<br>vertex (Vertex) |
| getCell(self, i) | Returns i-th cell (identified by cell number, not label).<br>**Returns**:<br>cell (Cell) |
| vertexLabel2Number(self, label) | Returns local vertex number corresponding to given label. If no label corresponds, throws an exception.<br>**Returns**:<br>vertex number (int) |
| cellLabel2Number(self, label) | Returns local cell number corresponding to a given label. If no label corresponds, it throws an exception.<br>**Returns**:<br>cell number (int) |
| getVerticesInBBox (self, bbox): | Returns the list of all vertices which are inside given bounding Box<br>**ARGS**:<br>- bbox (BoundingBox): bounding box |

| | |
|---|---|
| | **Returns:**<br>list of vertices inside bbox (list) |
| getCellsInBBox (self, bbox): | Returns the list of cells which bbox intersects with given bounding box<br>**ARGS:**<br>- bbox (BoundingBox): bounding box<br>**Returns:**<br>list of cells at least partially in bbox (list) |
| evaluateVertices(self, functor): | Returns the list of all vertices for which the functor is satisfied. The functor is a user defined class with two methods: *giveBBox*() which returns an initial functor bbox, and *evaluate* (obj) which should return true if functor is satisfied for a given object.<br>**ARGS:**<br>- functor: functor class<br>**Returns:**<br>list of all vertices for which the functor is satisfied (list) |
| evaluateCells(self, functor): | Returns the list of all cells for which the functor is satisfied. The functor is user defined class with two methods:*getBBox*() which returns an initial functor bbox, and *evaluate* (obj) which should return true if functor is satisfied for given object.<br>**ARGS:**<br>- functor: functor class<br>**Returns:**<br>list of all cells for which the functor is satisfied (list) |

# 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** |
|---|---|
| `__init__(self, mesh, number, label, vertices)` | Constructor. Creates the new cell.<br>**ARGS**:<br>- mesh(Mesh): the mesh to which cell belongs.<br>- number(int): local cell number<br>- label(int): cell label<br>- vertices(tuple): cell vertices (local numbers) |
| copy(self) | This will copy the receiver, making deep copy of all attributes EXCEPT mesh attribute<br>**Returns**:<br>the copy of receiver (Cell) |
| getVertices(self) | **Returns**:<br>the list of cell vertices (tuple of Vertex instances) |
| containsPoint(self, point) | **Returns**:<br>True if cell contains given point, False otherwise |
| getGeometryType(self) | **Returns**:<br>geometry type of receiver (CellGeometryType) |
| getBBox(self) | **Returns**:<br>bounding box of the receiver (BBox) |

# 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 |
| --- | --- |
| __init__(self, number, label, coords=None) | Constructor. Creates the new vertex instance.<br>**ARGS**:<br>- number(int): local vertex number<br>- label(int): vertex label<br>- coords(tuple): 3D position vector of vertex. |
| getCoordinates(self) | **Returns**:<br>receiver coordinates (tuple) |
| getNumber(self) | **Returns**:<br>receiver number (int) |
| getLabel(self) | **Returns**:<br>receiver label (int) |

# BoundingBox

Represents an axis aligned bounding box - a rectangle in 2d and a prism in 3d. Its geometry is described using two points - lover 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 |
| --- | --- |
| __init__(self, coords_ll, coords_ur) | Constructor. Creates the new Bounding box instance.<br>**ARGS**:<br>- coords_ll (tuple): coordinates of lower left corner<br>- coords_ur (tuple): coordinates of upper right corner |
| containsPoint (self, point) | Returns true if point inside receiver. |

| | |
|---|---|
| | **ARGS**:<br>- point (tuple): point coordinates<br>**Returns**:<br>True if point is inside receiver, false otherwise (Bool) |
| intersects (self, bbox) | **Returns**:<br>Returns true if receiver intersects given bounding box (Bool) |
| merge (self, entity) | Merges (expands) receiver with given entity (position or bbox)<br>**ARGS**:<br>- entity (tuple or BoundingBox): position vector (tuple) or bounding box.<br>**Returns**:<br>None |

# 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** |
|---|---|
| __init__(self,msg) | Constructor. Initializes the exception.<br>**ARGS**: |

| | |
|---|---|
| | - msg (string) Error message |
| __str__(self) | **Returns:** string representation of the exception, ie. error message (string). |

## Developing Application Program Interface (API)

In order to establish an interface between the platform and external application, one has to implement an Application 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 Application class API specification.

| Method | Description |
|---|---|
| __init__(self, file) | Constructor. Initializes the application. |
| getMesh (self, tstep) | Returns the computational mesh for given solution step. |
| getField(self, fieldID, time) | Returns the requested field at given time. Field is identified by fieldID. |
| setField(field) | Registers the given (remote) field in application. |
| getProperty(self, propID, time, objectID=0) | Returns property identified by its ID evaluated at given time. |
| setProperty(self, property, objectID=0) | Register given property in the application |
| setFunction(self, func,objectID=0) | Register given function in the application |
| solveStep(self, tstep) | Solves the problem for given time step. |
| finishStep(self, tstep) | Called after a global convergence within a time step. |

| getCriticalTimeStep() | Returns the actual critical time step increment. |
|---|---|
| getApplicationSignature() | Returns the application identification |
| terminate() | Terminates the application. |

Table 2: Application 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 Application 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 Application 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).

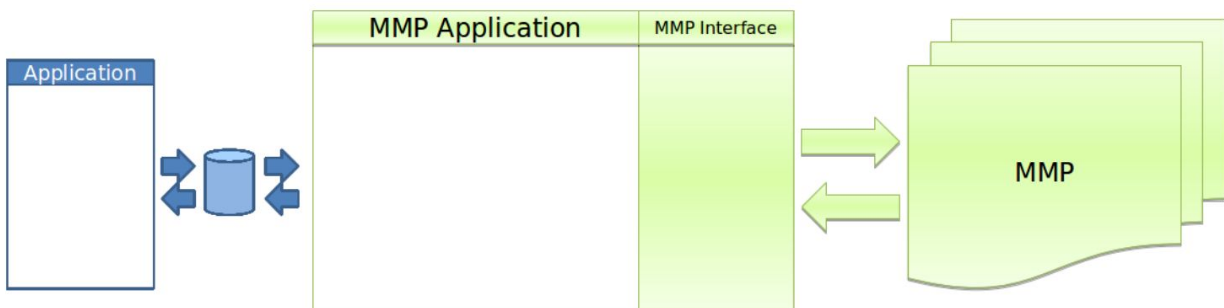



Fig. 2: Illustration of indirect approach

The example illustrating the indirect implementation is available from 2nd workshop material (located in examples/Workshop02/Demo31). 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. This three-step procedure is illustrated in the following example listing taken from Demo31. 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.

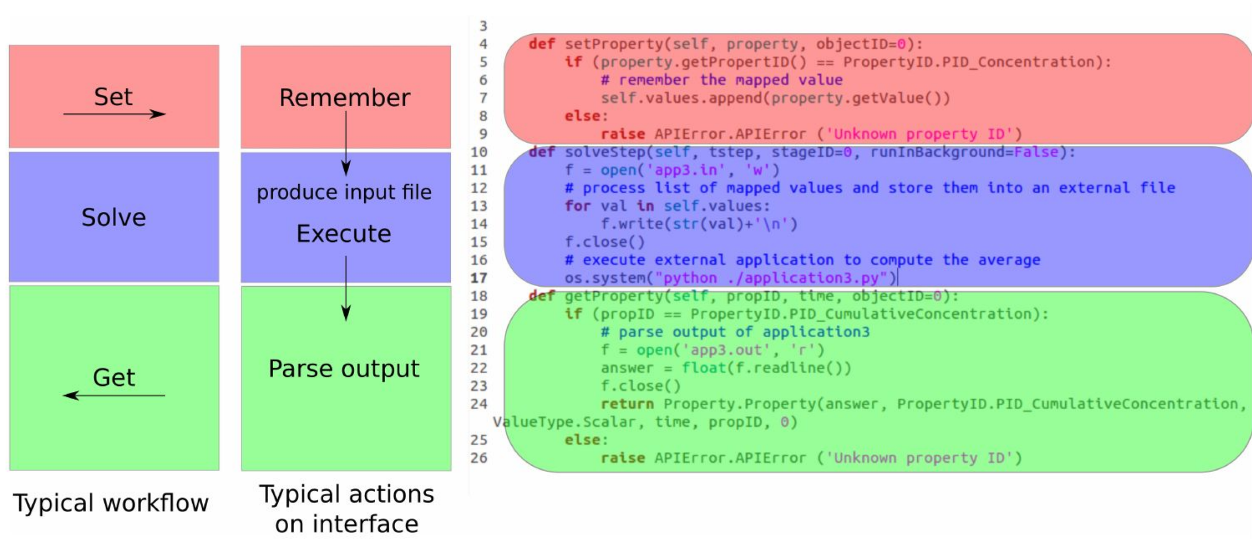



Fig. 3: Typical workflow in indirect approach to API implementation

# 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.

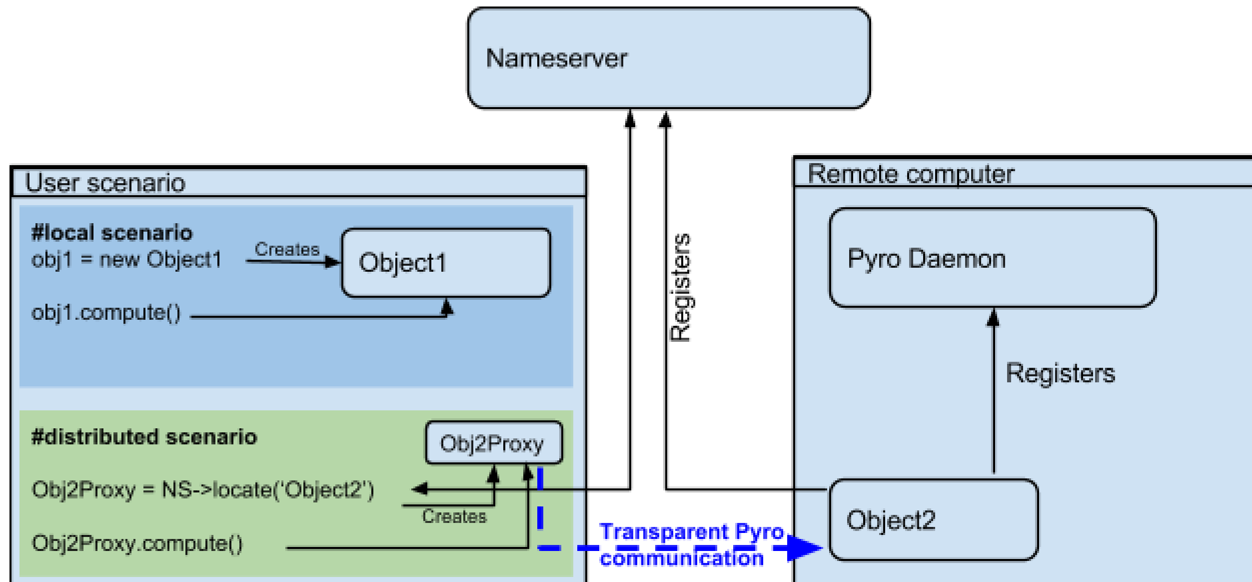



Fig.4: Local vs remote object communication scenarios compared

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 on Fig. 4.

## 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 involves 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)
```

When a secure communication over ssh is used, then typically 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 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 example10 for an example).

As an example, consider the simulation scenario composed of two applications running on two remote computers. 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. This situation is depicted in Fig. 5.

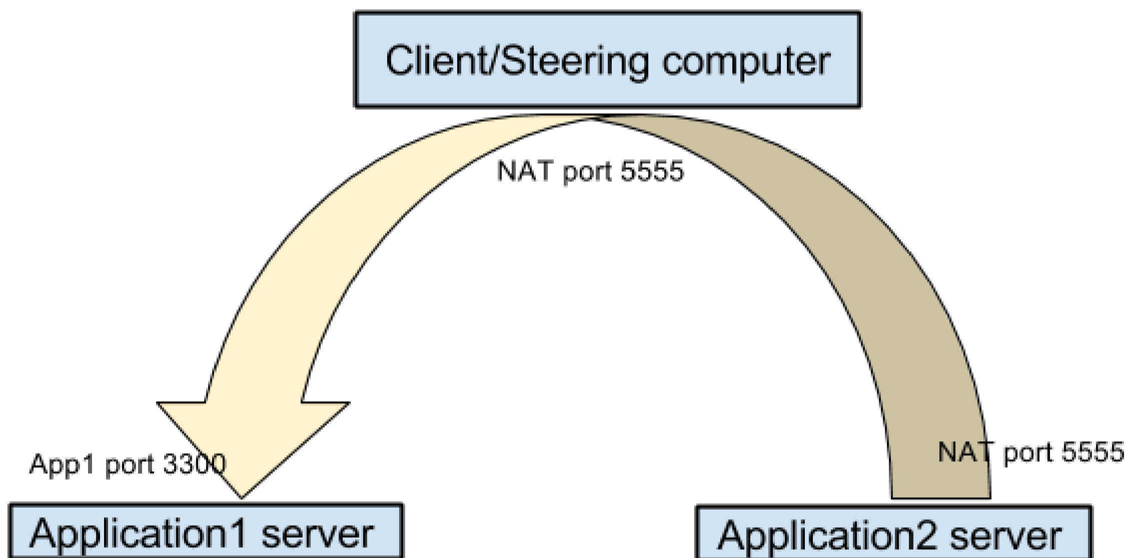



Fig. 5: Establishing a communication link between two application servers

## Requirements for distributed computing

Within the MMP project, the nameserver service is hosted at CTU infrastructure. For the use of the platform outside the MMP project a different Pyro nameserver should be set up and used, see Pyro documentation.

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. It setup does not requires 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.

## Internal platform solution - JobManager resource allocation

This solution has been developed from 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 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. 6 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.

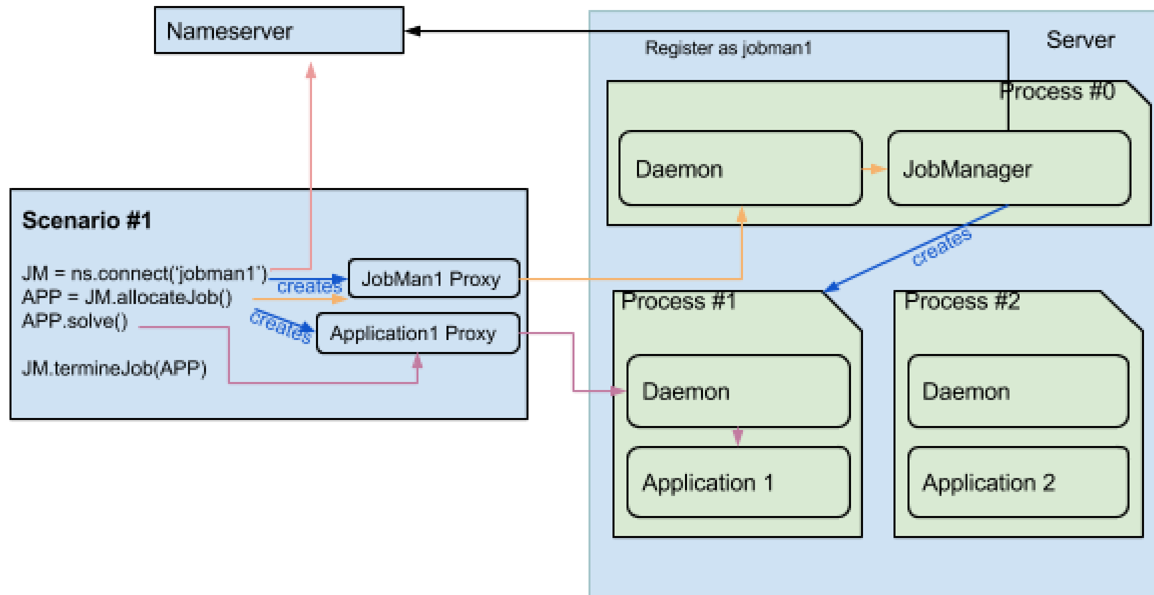



Fig. 6: Typical control flow with resource allocation using JobManager.

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. 7.

```
bp@jaja: /home/bp/Documents/projects/MMP/mupif.git/tools
bp@jaja: /home/bp/Documents/projects/MM...   bp@jaja: /home/bp/Documents/projects/MM...
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
```

Fig. 7: Screenshot of Job Manager monitoring tool

The user and access control is controlled using ssh authorization. The individual computational servers and their platform services are assumed to run behind a firewall. To establish the connection to a remote server and platform services a secure connection has to be established. This is realized using setting up ssh tunnel, that allows client to communicate with protected communication ports on the server, see Fig. 8. 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.

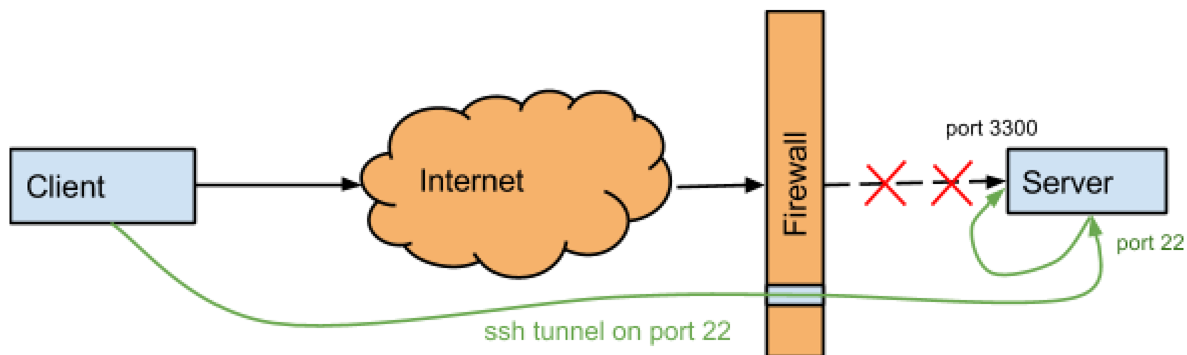



Fig. 8: Bypassing firewalls with ssh tunnels

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.

## Installation

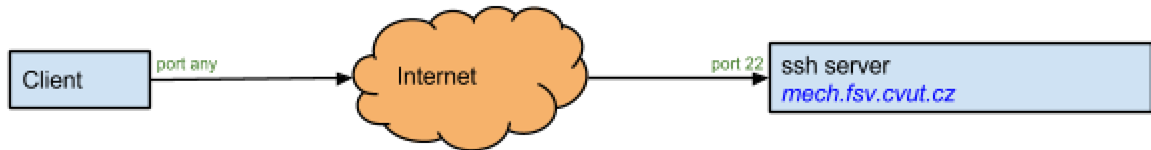
### Setting up ssh server

SSH server provides functionalities which generally allows to

- Securely transfer encrypted data / streams
- Securely transfer encrypted files (SFTP)
- 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 runs on a 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. 9 shows both cases.




Login via username/password:

Unix: *ssh mmp@mech.fsv.cvut.cz*
Windows: *putty.exe mmp@mech.fsv.cvut.cz*

Login via private/public keys:

Unix: *ssh mmp@mech.fsv.cvut.cz -i ~/project/keys/id_rsa*
Windows: *putty.exe mmp@mech.fsv.cvut.cz -i C:\Users\Keys\public-SSH2.ppk*

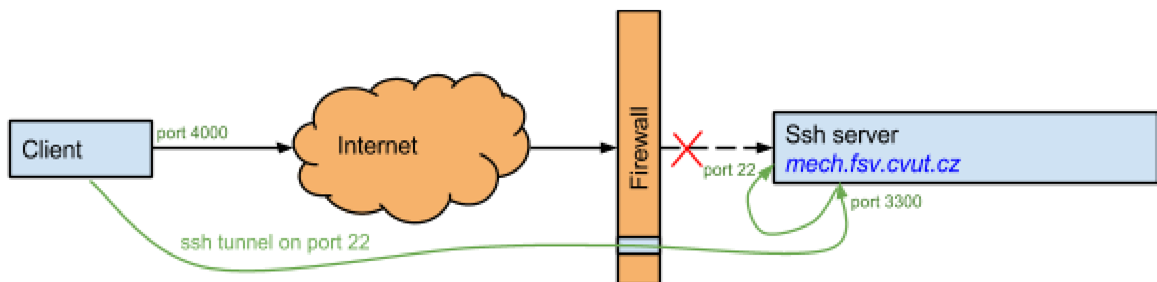
Add a public key to known hosts before the ssh connection

Fig. 9: 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 Figure 10, 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*

Fig. 10: Creating a ssh forward tunnel

## Setting up Job Manager

The skeleton for application server is distributed with the platform and is located in examples/Example06-JobMan. 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 conf.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 Section Platform installation. Also, the functional application API class is needed. Fig. 11 shows the flowchart

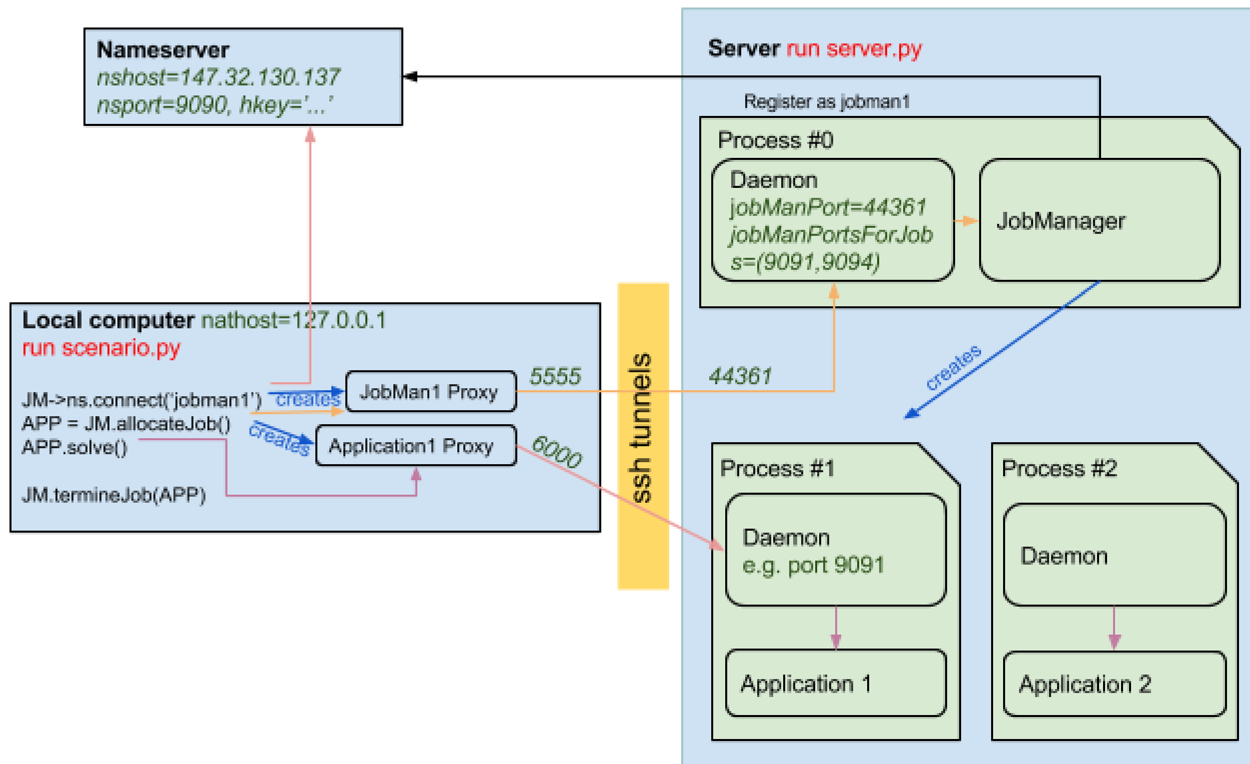



Fig. 11: *Example06-JobMan* displaying ports and tunnels in a distributed setup.

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/Example06-JobMan directory and customize settings in serverConfig.py.

### Configuration

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

| Variable | Description |
|---|---|
| deamonHost | hostname or IP address of the application server, i.e. daemonHost='147.32.130.137' |
| hostUserName | user name to establish ssh connection to server, i.e. hostUserName='mmp' |
| jobManPort | Server port where job manager daemon listens, i.e., jobManPort=44361. |
| jobManNatport | Port reported by nameserver used to establish tunnel to destination JobManager port (jobManPort), i.e. jobManNatport=5555 |
| jobManName | Name used to register jobManager at nameserver, i.e, jobManName='Mupif.JobManager@micress' |
| jobManPortsForJobs | 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 relesead immediately by operating system, see jobManMaxJobs)<br>Example:<br>jobManPortsForJobs=( 9091, 9092, 9093, 9094) |
| jobManMaxJobs | Maximum number of jobs that can be running at the same time.<br>jobManMaxJobs=4 |
| 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. |

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 application server run:

```
$ python 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 following command:

```
$ python jobManStatus.py -j Mupif.JobManager@demo -h 147.32.130.137 -u mmp -p 44361 -n 147.32.130.137 -r 9090 -k mmp-secret-key -t
```

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. 12.

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.

```
bp@jaja: /home/bp/Documents/projects/MMP/mupif.git/tools
bp@jaja:~/Documents/projects/MMP/mupif.git/tools$ python jobManTest.py -j Mupif.JobManager@demo
 -h 147.32.130.137 -u mmp -p 44361 -n 147.32.130.137 -r 9090 -k mmp-secret-key -t
hkey:mmp-secret-key
Nameserver:147.32.130.137:9090
JobManager:Mupif.JobManager@demo@147.32.130.137:44361
Jobmanager uri:PYRO:obj_7b899f9f6437412bb3a4a9195ec24149@127.0.0.1:5555
Application 16@Mupif.PingServerApplication successfully allocted
Terminating 16@Mupif.PingServerApplication
Time consumed 2.272660 s
bp@jaja:~/Documents/projects/MMP/mupif.git/tools$
```

Fig. 12: Testing job manager in a simple setup

Troubleshooting

- 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*

### Simple distributed example using jobManager resource allocation

The process of allocating a new instance of remote application involves several steps, see Table 3. First, the secure connection to corresponding job manager has to be established using ssh tunnel. In the second step, the jobManager is requested to allocate a new application instance and returns corresponding URI of new application. As the application is executed in a separate process, a second secure connection to the new process pyro daemon has to be established and the proxy of application instance obtained. When the scenario is terminating, all these connections have to be correctly terminated. As this involves a lot of steps, a utility function *PyroUtil.allocateApplicationWithJobManager* is provided, returning an instance of RemoteAppRecord class, which encapsulate all the details of opened connections, etc. It provides two useful methods: *getApplication()* returning application Proxy and *terminate()* that can be used to correctly terminate the application and close all connections. Here we show again the example presented in section Platform operations, with the potential modifications for the distributed case shown in blue color. Note that the differences are only in the setup and

terminating part, the core logic of the scenario remains the same for local as well as distributed case.

```
from mupif import *
import application1
import application2

time  = 0
timestepnumber=0
targetTime = 10.0

#locate nameserver
ns = PyroUtil.connectNameServer(nshost=conf.nshost, nsport=conf.nsport,
hkey=conf.hkey)
#establish secure tunnel to JobManager running on (remote) server
try:
       app1Rec = PyroUtil.allocateApplicationWithJobManager (ns, conf.app1JobManRec,
                                                           conf.jobNatPorts.pop())
       app2Rec = PyroUtil.allocateApplicationWithJobManager (ns, conf.app2JobManRec,
                                                           conf.jobNatPorts.pop())

       app1 = app1Rec.getApplication()
       app2 = app2Rec.getApplication()


except Exception as e:
       logger.exception(e)
       app1Rec.terminate()
       app2Rec.terminate()
       break

#establish secure tunnel to JobManager running on (remote) server
try:
       jobMan2 = PyroUtil.connectApp(ns, conf.jobManName)
       retRec2 = jobMan.allocateJob(PyroUtil.getUserInfo(), natPort=conf.jobNatPort)
       app2 = PyroUtil.connectApp(ns, retRec2[1])
except Exeption as e:
       logger.error("Following API error occurred: %s" % e )
       break


# 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
```

```
        print ("Step: %d %f %f" % (timestepnumber, time, dt) )
        # 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 ("Result: %f" % prop.getValue() )


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

# terminate
app1Rec.terminate()
app2Rec.terminate()
```

Table 3: Simple example illustrating simulation scenario

# Acknowledgements

The development of MuPIF has been funded by Grant Agency of the Czech Republic - Projects No. P105/10/1402.
The development is at present supported by the EU project Multiscale Modelling Platform: Smart design of nano-enabled products in green technologies, project number 604279.

# 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 2.7. 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 **15 th International Conference on Civil, Structural, and Environmental Engineering Computing**, 1st - 4th of September 2015, Prague (Czech Republic).