On PDE problem solving environments for multidomain multiphysics problems

Christos Antonopoulos, Manolis Maroudas, and Manolis Vavalis

University of Thessaly, Department of Electrical and Computer Engineering,
Gklavani 37, 38221 Volos, Greece
{cda,emmmarou,mav}@uth.gr

Abstract. This paper presents the design, the prototype implementation and the preliminary evaluation of an enhanced meta-computing environment based on the FEniCS Project and focused on multi-domain multi-physics problems modeled with partial differential equations. It is based on scripting languages and their practices, and on the Service Oriented Architecture paradigm and the associated web services technologies. Our design is generic, covering a wide range of problems but our proof of concept implementation is restricted to elliptic PDEs in two or three dimensions.

Keywords: problem solving environments, numerical solution of PDEs, scientific high performance meta-computing, numerical software

1 Introduction

Advances in hardware and software technologies in the 1980s led to the modern era of scientific modeling and simulation. This era seems to come to an end. The simulation needs in both industry and academia mismatch with the existing software platforms and practices, which to a great extent have remained unchanged for the past several decades. We foresee that this mismatch, together with the emerging ICT advances and the cultural changes in scientific approaches will lead to a new generation of modeling and simulation.

This paper proposes approaches for designing, analyzing, implementing and evaluating new simulation frameworks particularly suited to multi-domain and multi-physics (MDMP) problems that have Partial Differential Equations (PDEs) in their foundations. These types of problems appear frequently on real world problems. Considering also their heavy computational needs it seems reasonable to make them more accessible to the programmer while reducing their execution time using every available device/machine on a system/network.

We focus on designing a software platform that facilitates the numerical solution of PDEs associated with MDMP mathematical models. In particular, we propose an enhanced meta-computing environment which is based on: (a) scripting languages (Python) and their practices and (b) on the Service Oriented Architecture (SOA) paradigm and the associated web services technologies.

Although our design is generic, covering a wide range of problems, our proof of concept implementation is restricted to elliptic PDEs in two or three dimensions. Furthermore, we show that that our approach can easily exploit state of the art meta-computing methodologies (Schwartz splitting, hybrid stochastic deterministic methods, ...), numerical solvers (finite element modules from deal.II, interpolants, ...) and modern computer architectures. Specifically, it clearly shows that our tool can easily exploit state of the art numerical solvers including those available in FEniCS [6] and deal.II [2], domain decomposition methods with or without overlapping [5] [12] Monte Carlo based hybrid solvers [10], rectangular or curvilinear domains and interfaces and beyond.

2 Meta-computing algorithms for MDMP problems

Traditional linear PDE solvers follow a simple workflow. We first discretize the problem (domain and derivatives), and then solve the resulting linear algebraic problem. Unfortunately, this approach is not best suited for MDMP problems since it leads to monolithic PDE solvers that treat the MDMP problem as a coherent all that does not exploit the "multi" nature of the problem. For such problems, these solvers are expensive to develop, difficult to maintain and reuse. Their mapping to multi-processing machines is rather challenging.

Meta computing algorithms provide an attractive alternative. They allow us to exploit the problem characteristics and view its solution process as a workflow that involves individual, relatively simpler PDE solvers that are associated with the multi nature of the problem and can be fine tuned and easily mapped through high level parallelism.

2.1 Hybrid, deterministic-stochastic methods

The Monte Carlo method has the capability to provide approximate solutions to a variety of mathematical problems, not necessarily with probabilistic content or structure, by performing statistical sampling experiments. About a century has been passed since the discovery of methods which based on the Monte Carlo concept provide numerical approximations to PDE problems. These methods generate random numbers and by observing certain of their characteristics and behavior are capable of calculating approximations to PDE solutions.

It has been proposed (see [10] and references within) that the above stochastic solver can be combined with traditional PDE solvers to develop a hybrid solver that enjoy several very desirable characteristics in many respects.

Given the overall domain Ω with boundary $\vartheta\Omega$, the associated PDE and a particular domain of interest $D \subset \Omega$ with boundary $\Gamma = \vartheta D \backslash \vartheta\Omega$, the main steps of a hybrid stochastic/deterministic solver are:

Stochastic pre-processing: Monte Carlo-based walks on spheres inside Ω to compute an approximation of the solution at selected points on D.

Interpolation: Interpolation procedures which using the computed in the previous step solutions on particular points on D constructs the interpolant of the solution on D which acts as a boundary conditions for the local PDE sub-problems.

Deterministic solving: Solves each one of the independent local PDE sub-problems generated by the above decoupling of the original PDE problem. Selected a local conventional solver for each resulting sub-problems that is of our interest and compute the solution.

Our prototype implementation concentrates on the Poisson equation, narrowed on the unit square or unit cube for 2D and 3D problems respectively. It utilizes high, quality state of the art software components that include finite solvers from the deal.II [2] library for the deterministic solving step, 2D and 3D interpolants, plot and visualization modules etc..

2.2 Domain Decomposition Methods

The classical Schwarz alternating procedure demonstrates the basic mathematical idea of overlapping domain decomposition methods. These methods [5], are efficient, flexible and best suited for MDMP problems. Several of these methods are inherently suitable for parallel computing the solution of PDEs. They all offer a reasonable alternative since they are based on a physical decomposition of a global MDMP problem. The global solution is then sought by solving the smaller subdomain problems collaboratively and then combining their individual solutions.

Let us consider an example consisting of the domain $\Omega = \Omega_1 \cup \Omega_2$ with perhaps different elliptic operators on each subdomain. Γ_i is the internal boundary of subdomain $\Omega_i, i = 1, 2$.

Schwarz methods are realized through the following iterative procedure for finding the approximate solution in the entire composite domain Ω. Let u_i^n denote the approximate solution in subdomain Ω_i, and f_i denote the restriction of f to Ω_i. Starting with an initial guess u_0, we iterate for n = 1, 2, ... to find successive approximate solutions $u^k, k = 1, 2, \ldots$. Assuming, without loss of generality in our approach, that the original MDMP problem consists of the Poison equation in Ω with homogeneous Dirichlet conditions on Ω we iterate between the two sub-problems as follows:

$$\begin{aligned} -\nabla^2 u_1^n &= f_1 \;\; in \;\; \Omega_1 & \qquad -\nabla^2 u_2^n &= f_2 \;\; in \;\; \Omega_2 \\ u_1^n &= g \;\; on \;\; \vartheta\Omega_1 \backslash \Gamma_1, & u_2^n &= g \;\; on \;\; \vartheta\Omega_2 \backslash \Gamma_2, \\ u_1^n &= u_2^{n-1} \mid \Gamma_1 \;\; on \;\; \Gamma_1 & u_2^n &= u_1^n | \Gamma_2 \;\; on \;\; \Gamma_2. \end{aligned} \tag{1}$$

Within each iteration, the two problems continuously update the internal Dirichlet conditions on Γ_1 and Γ_2. Note that the classical alternating Schwartz methods usually have limited parallelism. There exist variants of these methods (eg additive Schwarz methods) that inherently promote parallel computing and

although their rate of convergence is lower, the associated iteration scheme is inherently parallel.

Interface relaxation methods [12] are essentially non-overlapping domain decomposition methods that follow the iterative structure of the Schwartz method but in a more complicate manner. Besides that, in our meta-computing implementation framework these methods can be treated in a completely similar manner that due to space limitation will not be presented here.

3 FEniCS Extensions for MDMP PDE Problems

The FEniCS project [6] is an open-source collection and integration of software tools specialized on automated, high quality and high performance solution of differential equations.

The main user interface of FEniCS is Dolfin [7], a C++ and Python library. It provides a problem solving environment for models based on PDEs. It implements core parts of the functionality of FEniCS, including data structures and algorithms for computational meshes and finite element assembly. It also wraps the functionality of other FEniCS components and external software, and is responsible for the correct communication between them.

FEniCS targets user-friendly notation and support for rapid development. It supports weak formulations for the representation of PDEs through the Unified Form Language (UFL) [1]. UFL is integrated with Dolphin and defines a flexible user interface for defining finite element spaces and expressions for weak forms in a notation close to mathematical notation. It can handle complicated equations efficiently. Differentiation of expressions and forms are also integrated in the language.

The main goal of our extensions is to design and implement an open, enhanced meta-computing environment supporting MDMP problems, without changing the back-end of FEniCS (problem assembly, linear algebra solvers etc.). Our platform utilizes and extends the Python user interface of Dolfin, as Python syntax is closer to UFL, being at the same time fitter for rapid prototyping. Support for multi-domain multi-physics (MDMP) problems is implemented on top of the existing functionality, either as new Python modules using the available data structures and classes, or as external dynamically shared C++ libraries, wrapped as Python modules using SWIG [3].

3.1 Extensions for MDMP PDE Problems with Overlapping Domains

We implement the additive Schwarz method and use it as a high level solver for MDMP problems with overlapping domains. The geometry, interfaces, discretization, boundary values and equations applicable on each subdomain are described using UFL and Dolphin in a separate file per subdomain. This organization treats different subdomains as distinct programming units, facilitating the parallel or distributed solution of the problem on different subdomains. This

is particularly helpful in case web services are used, as discussed in Section 4. All datatypes used are either pure Python or FEniCS objects. There is no dependence from third party software libraries at this level.

Each subdomain object must override a number of methods implicitly called before each invocation of the solver.

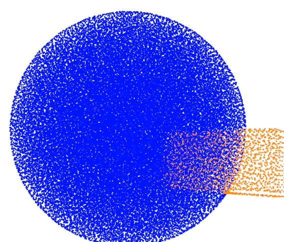
init() This method holds the UFL [1] definition of the subdomain and sets as class attributes the subdomain's function space, linear and bi-linear form of the PDE.

neighbors() It provides information to the solver about the other subdomains this subdomain overlaps with, in order for the solver to automatically update the interface values after each iteration.

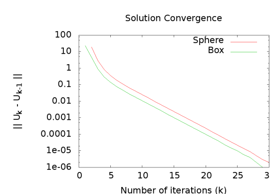
boundaries() It informs the solver about the fixed external boundaries of the subdomain.

The entry point of the iterative solver is the solve() method. It takes as arguments an object with the configuration of the solving environment (max iterations, tolerance, etc) and a Python list of user defined problem objects.

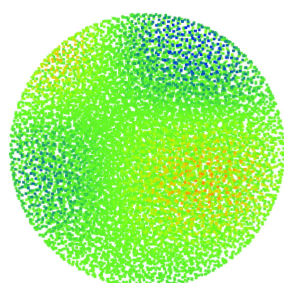
After each iteration, for each subdomain solution, the algorithm checks a set of termination criteria evaluating convergence or whether a maximum number of iterations has been reached.



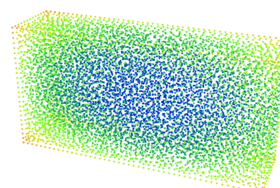
(a) Subdomains topology



(b) Convergence rate for the two subproblems



(c) Solution for the sphere subdomain



(d) Solution for the box subdomain

Fig. 1. A sample 3D problem with two overlapping subdomains

As an example, assume a 3D problem with two subdomains: a sphere and a box that overlap as shown in figure 1(a). Figures 1(c) and 1(d) depict the solution using the iterative Schwarz solver, whereas figure 1(b) depicts the convergence rate for the two subdomains until reaching the user-specified accuracy.

Listing 1.1 outlines the definition of the box subdomain (box3D_1.py) on top of a common skeleton file. The definition of the sphere subdomain (sphere3D_1.py) is similar. Listing 1.2 shows the code that, given the subdomain definitions, drives the iterative solver. The user-required changes on the skeleton and driver are in bold. The subdomains can be – and are in the example – configured with different meshes, discretizations and PDEs. Note also that using a remote solver would be completely straightforward, by substituting line 7 with the commented-out lines 5-6.

Listing 1.1. Box subdomain definition based on a common skeleton.

```
1   # user defined methods
2   def OveralappingWithOther(): pass
3   def userDefinedUFL(): pass
4   def userDefinedBoundaryCondition(): pass
5
6   # skeleton example
7   def ExtBC(x,on_boundary):
8       return on_boundary and not OveralappingWithOther()
9
10  def ExtIface(x,on_boundary):
11      return on_boundary and OveralappingWithOther()
12
13  class Problem(ConfigCommonProblem):
14          # Override the API methods init(), neighbors() and boundaries()
15
16          def init(self,*args,**kwargs):
17          self.domain_name = 'box'
18          mesh = Mesh(Box(-2,-1,-.5,2,1,.5),128)
19          self.V = FunctionSpace(mesh,'Lagrange',1)
20          self.a, self.L = userDefinedUFL(V)
21
22      # return a dictionary with interfaces for each neighbor
23      def neighbors(self):
24          interface = {}
25          interface[self.domain_name] = ExtIface
26          return interface
27
28          # return a list with all the external boundaries
29      def boundaries(self):
30          bc = DirichletBC(self.V, userDefinedBoundaryCondition(), ExtBC)
31          return [ bc ]
```

Listing 1.2. Code solving a 3D problem with two overlapping subdomains.

```
1   from dolfin import *
2   import solverconfig
3   import solver
4
5   # client = hmc.RemoteClient(wsdl_url)
6   # client.set_options(timeout=timeout_in_seconds)
7   client = hmc.LocalClient()
8
9   import sphere3D_1 as sphere
10  import box3D_1 as box
11
12  sp = sphere.Problem(client=client)
13  bp = box.Problem(client=client)
14  subdomains=[ sp, bp ]
15
16  config = solverconfig.Config3D()
17  solver.solve(subdomains=subdomains, config=config)
```

3.2 Hybrid, Deterministic-Stochastic Method Extensions

We introduce extensions to FEniCS to support hybrid deterministic-stochastic methods. More specifically, a stochastic step is used to evaluate values at interfaces, whereas default FEniCS support can be used for interpolation and solving.

This design decouples the stochastic interface estimation from the actual solving and allows it to be implemented on any device (including CPUs, GPUs or even FPGAs) in order to take advantage of the vast parallelism inherently available in Monte-Carlo methods. Our implementation includes a POSIX threads version for CPUs, as well as an OpenCL [11] version for any OpenCL-capable device (including CPUs and GPUs).

The functionality is available in Dolphin in the form of an MC class, offering the MC::montecarlo() method. The latter takes a description of the interface as input and outputs estimations for the values on the interface. Both 2D and 3D problems are supported.

The montecarlo() method takes the same arguments with the DirichletBC class of FEniCS, plus a description of the original domain and the subdomain of interest. Using the DirichletBC methods we obtain the points on the interface and call the new montecarlo() method for them. The call returns the estimated values of all interface points (nodes) assigned to a new DirichletBC object. The latter can then be used anywhere in the rest of the program.

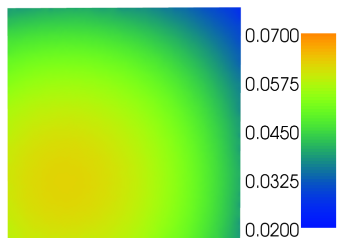
Listing 1.3 outlines and example of using the stochastic support introduced in FEniCS to solve a PDE in an internal, rectangular subdomain of the original domain, by stochastically estimating the values at the interface of the internal subdomain. Once again, changes introduced by our extensions are highlighted in bold.

Listing 1.3. Example of montecarlo() method in user code.

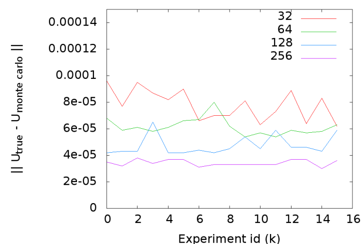
```
1   from dolfin import *
2   import hybridmc as hmc  # the platform's Python module
3
4   def onbc(x,on_boundary):
5       return on_boundary
6
7   def mc_test_2D(Omega, Subdomain):
8       x, y = variable(Expression("x[0]")), variable(Expression("x[1]"))
9       expr = (x)*(x-1)*(y)*(y-1)
10
11      mesh = Mesh(SubDomain,128)
12      V = FunctionSpace(mesh,'Lagrange',1)
13
14      u, v = TrialFunction(V), TestFunction(V)
15      f = -Laplacian(expr,x,y)
16      a = inner(grad(u), grad(v))*dx
17      L = f*v*dx
18
19      # get expressions as strings
20      f_expr, q_expr = hmc.tools.cppcode(expr,x,y), hmc.tools.cppcode(f,x,y)
21      mcbc, est = client.montecarlo(V, onbc, OpenCL=True, Omega=Omega,
22                                    f=f_expr, q=q_expr)
23      sol_mc = Function(V)
24      solve(a==L,sol_mc,[ mcbc ])
25
26  if __name__ == '__main__':
27      Omega2D = [ 1., 1. ]
28      SubDomain2D = Rectangle(.4, .8, .4, .8)
29      client = hmc.LocalClient()
30      mc_test_2D(Omega2D, Subdomain2D)
```

The client object at line 29 above encapsulates the local/remote functionality of the method. We discuss more about web services and client objects in chapter 4.

Figure 2 depicts the solution provided by the hybrid stochastic/deterministic Monte Carlo-based solver for the sample problem of listing 1.3, as well as the absolute error with respect to a fully deterministic solver for a set of 16 experiments.



(a) Solution of a Monte Carlo-based solver



(b) Solution error w.r.t. a fully deterministic solver. Different lines correspond to different mesh resolutions

Fig. 2. Hybrid solution and error estimation with respect to a deterministic solver for the sample problem

4 Web Services Layer

The scientific community has embraced the Web, enabling researchers as well as practitioners to closely collaborate and share resources. This resulted primarily in the publication of information while the availability of computational services has been rather limited and to a great extend monolithic, mostly in the form of e-science platforms that are expensive to build and difficult to reuse outside their scope and environment.

We envision a radically new way of deploying, publishing, sharing, discovering and re-using Scientific Computing resources in every day practice. For that we argue the necessity of an open, balanced and ever-evolving ecosystem of web services that:

- relieves consumers from the pain of selecting/installing/running the most appropriate algorithm/software/machine components for their scientific computing needs.
- allows producers to offer their scientific computing components in an easy to be discovered/packaged/consumed way.
- enables computing facilities to accommodate a wide range of consumers and producers in an open, dynamic, and value adding manner.
- advances the science of scientific computing towards problem solving with the optimum available algorithm/implementation/machine combination

In our study we explore the idea of having the ability to develop, offer and consume MDMP related computational modules in a transparent and abstract way within our above described platform and through the Web. For this we enhance our platform with a web service layer utilizing the following XML based standards. For detailed information about Web services in general and the above standards in particular please see [9].

SOAP: (Simple Object Access Protocol) for exchanging structured information within web services in computer networks. It relies on other application layer protocols, such as HTTP for message negotiation and transmission.

WSDL: (Web Services Description Language) used for describing the functionality a web service offers, how it can be called, what parameters it expects, and what data structures it returns.

ebXML: (e-business XML) that allows web service providers to publish their services and consumers to query for service availability and description.

The overall scenario for developing a MDMP solver under the SOA paradigm is the following. We first wrap up any of the software modules mentioned above (or any other related legacy code) as a web service and publish it on our ebXML directory utilizing any of the available IDEs or platforms. This may be accomplished through one of the several available Integrated Developing Environments (IDE) or platforms. In particular we have implemented the above task in several different ways, using WSO2, .Net, Eclipse, or Spyne (see details given below) in a systematic manner that makes wrapping and deployment a more or less routine procedure with no particular challenges. Next, any developer or even any software agent could query to ebXML for particular services, receive the list of available ones, select the most appropriate and bind to it automatically through its WSDL file even at run-time.

For a particular wrapper's implementations we have selected Spyne [8] one out of the many existing Python frameworks and briefly describe our main steps below. Listing 1.4 shows a simple server function definition that wraps the Monte Carlo method. The deployment of the server code in listing 1.4 can be done as shown in listing 1.7. The RemoteClient class (see below) utilizes Suds [4], a lightweight SOAP-based web service client for Python which reads WSDL files at runtime. Upon creation, it parses the WSDL and derives from it a representation which is used to provide the user with a service description for message/reply processing.

In order to have a consistent API between local and remote methods, apart from the RemoteClient class, the platform also defines a LocalClient class, as shown below, with the same API methods. In the case of the RemoteClient, the input data are sent to the remote server which in turn responds with the output data. Listing 1.5 shows the base definition of the RemoteClient and LocalClient classes. Any underlying implementation differences are transparent to the user who in both cases receives the result the conventional local function call way, as shown in listing 1.6.

5 Conclusion

We design a meta-computing platform to target MDMP problems, modeled with PDEs, based on (but not limited to) the FEniCS project. The platform's environment provides a high level scripting API in Python, that utilizes the FEniCS UFL domain specific language.

Our environment allows domain experts to focus on expressing the models than delving into implementation details, programmers to effectively select the most appropriate available software module for a particular component (subdomain) of the problem with respect to its associated single physics model and

users to efficiently deploy and run MDMP computations on loosely coupled distributed and heterogeneous compute engines.

We also show how to integrate remote functionality from machines over the web in a consistent and transparent way to the end user, following widely accepted standards. Our generic design allows us to exploit state of the art software libraries and explore new solving approaches for MDMP problems. It essentially allow us to replace our traditional software library based viewpoint with and the SOA based one that aggressively promote meta-computing and software reuse at large.

6 Acknowledgment

This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) Research Funding Program: THALES. Investing in knowledge society through the European Social Fund.

References

1. M. Alnæs. *UFL: a Finite Element Form Language*, chapter 17. Springer, 2012.
2. W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007.
3. D. Beazley. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, TCLTK'96, pages 15–15, Berkeley, CA, USA, 1996.
4. Fedorahosted.org. Suds is a lightweight soap python client for consuming web services., 2014. [Online; accessed 23-October-2014].
5. M. J. Gander. Schwarz methods over the course of time. *Electronic Transactions on Numerical Analysis*, pages 228–255, 2008.
6. A. Logg, K. Mardal, G. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.
7. A. Logg and G. Wells. DOLFIN: Automated Finite Element Computing. *ACM Transactions on Mathematical Software*, 37(2), 2010.
8. Arskom Ltd. spyne - rpc that doesn't break your back., 2014. [Online; accessed 23-October-2014].
9. N. Papazoglou. *Web Services: Principles and Technology*. Pearson Prentice Hall, 2008.
10. G. Sarailidis and M. Vavalis. Implementing hybrid PDE solvers, 2013. http://dx.doi.org/10.6084/m9.figshare.1134520.
11. J. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.
12. P. Tsompanopoulou and E. Vavalis. An experimental study of interface relaxation methods for composite elliptic differential equations. *Applied Mathematical Modelling*, 32(8):1620 – 1641, 2008.

Appendix

Listing 1.4. A simple server function definition.

```
1  from spyne import Application, rpc, ServiceBase, Integer, Double, Array
2  from spyne.protocol.soap import Soap11
3  import _hybridmc as core
4  import numpy as np
5
6  class MDMPService(ServiceBase):
7      """ 1. convert the input Python lists to numpy arrays
8          2. call the core method and return output as Python list """
9      @rpc(Array(Double), Integer, Array(Double), Integer, String, String, Boolean,
10         _returns=Array(Double))
11     def montecarlo(ctx, dims, dim, coords, nof_nodes, f, q, OpenCL):
12         D = np.array(dims, dtype=np.float_)
13         node_coord = np.array(coords, dtype=np.float_)
14         if not OpenCL:
15             f = Expression(f)
16             q = Expression(q)
17         return core.montecarlo(D, dim, node_coord, nof_nodes, f, q)
```

Listing 1.5. Base definition of RemoteClient and LocalClient.

```
1  from suds.client import Client
2
3  class RemoteClient(Client):
4      def __init__(self, *args, **kwargs):
5          self.is_local = False
6          Client.__init__(self, *args, **kwargs)
7
8  class LocalClient():
9      def __init__(self, *args, **kwargs):
10         self.is_local = True
```

Listing 1.6. Create client objects from user-code.

```
1  from dolfin import *
2  import hybridmc as hmc
3
4  if use_remote_client:
5      client = hmc.RemoteClient(wsdl_url)
6      client.set_options(timeout=90) # ...
7  else:
8      client = hmc.LocalClient()
```

Listing 1.7. Server deployment.

```
1  from spyne import Application
2  from spyne.server.wsgi import WsgiApplication
3  from wsgiref.simple_server import make_server
4  from mdmp_service import MDMPService
5  import logging
6
7  app = Application([MDMPService], 'spyne.examples.hello.soap',
8                    in_protocol=Soap11(validator='lxml'), out_protocol=Soap11())
9  wsgi_app = WsgiApplication(app)
10 # logging code ommited
11 server = make_server('127.0.0.1', 8000, wsgi_app).serve_forever()
```