Submitted to *Management Science*
manuscript

Optimization Services: A Framework for Distributed Optimization

Robert Fourer
Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, Illinois 60208, USA, 4er@iems.northwestern.edu

Jun Ma
Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, Illinois 60208, USA, maj@iems.northwestern.edu

Kipp Martin
Graduate School of Business,University of Chicago, 5807 South Woodlawn Avnenue, Chicaogo, Illinois 60637, USA, kipp.martin@chicagogsb.edu gsbkip.chicagogsb.edu

This paper describes the Optimization Services (OS) project. The OS project provides a framework consisting of standards for 1) representing optimization instances, results, and solver options, 2) communication between clients and solvers, and 3) the discovery and registration of optimization-related services in a distributed environment using Web Services. The ultimate goal of this project is to provide an environment where optimization system components, including modeling language environments, servers, registries, communication agents, interfaces, analyzers, solvers, and simulation engines, can be implemented as services under a unified framework so that customers can use these computational services much like utilities. We also describe a reference implementation that is a COIN-OR project.

Key words: optimization, modeling languages, distributed computing, XML, Web Services, COIN-OR
History: This paper was submitted February 1, 2008.

1. Introduction

A key objective of the operations research (OR) community is to promote the use of optimization. In order to achieve this objective, it must be easy to integrate optimization tools into a modern corporate IT (information technology) infrastructure. The difficulty in achieving this is that currently the OR community has its standalone tools like modeling languages and solvers designed to work on a single machine, while the IT community is moving to tools like XML, Service Oriented Architectures, and Web Services that facilitate distributed computing, and the two are not designed to work with each other. The OR community could much more easily achieve its objective if optimization tools were built into technologies the IT community was already using. Witness the success of Excel Solver – the OR community got that one right. The solver engine was seamlessly integrated within an existing and ubiquitous business tool. Now that the IT community is moving in the direction of distributed computing, the OR community must follow this lead and seamlessly integrate optimization tools into a distributed computing environment.

Current key IT technologies include:

1. Extensible Markup Language (XML) for Data
2. Service Oriented Architecture
3. Web Services

Corollary 1. *The OR community* **must** *use these technologies when building and solving optimization models in order to integrate optimization capability into a modern corporate IT infrastructure.*

The technologies mentioned above have facilitated the increasing prevalence of *software as a service.* By software as a service, we mean software residing on a server that is accessed by a client machine over a network as opposed to all of the software residing on the client machine. Examples of software as a service include CRM (customer relationship software, see, e.g. `salesforce.com`), tax preparation, gmail, and Google Calendar. All of the major players in software are promising software as a service. The trend is clearly moving away from the fat client loaded with lots of heavyweight applications, and towards distributed computing.

Corollary 2. *Optimization should be available as a software service. It should be easy to solve optimization problems of any type (linear, integer, nonlinear, stochastic, etc), at any time, if you are hooked up to the network.*

Optimization Services is our attempt to make optimization a software service. Optimization Services is needed because:

• There are numerous modeling languages each with its own format for storing the underlying model instance. As such there is no standard for representing problem instances, especially nonlinear optimization instances. Optimization Services provides a general and robust format for representing general optimization instances.

- There are numerous solvers each with its own application program interface (API). There is no standard API. Optimization Services provides a common solver interface with `get()`, `set()`, and `calculate()` methods.
- The variety of operating systems, chip architectures, and compilers makes it difficult for vendors of optimization software to support every platform. Optimizations Services provides communication protocols that allow a client machine of any platform type to communicate with a server solver of any platform type.
- There is no standard for representing optimization results or solver options. Optimization Services provides these standards.
- There is no standard protocol for solvers to register their service over a network or for clients to discover a solver service over a network. Optimization Services provides registry and discovery protocols.

The Optimization Services project is a *framework* for distributed optimization. By framework, we mean a set of standards (protocols) for 1) representing optimization instances, results, and solver options, 2) communication between clients and solvers, and 3) the discovery and registration of optimization-related services in a distributed environment using Web Services.

The ultimate goal of this project is for all optimization system components, including modeling language environments, servers, registries, communication agents, interfaces, analyzers, solvers, and simulation engines, to be implemented as services under the OS framework, and for customers to use these computational services much like utilities, with no specialized knowledge of optimization algorithms, problem types, and solver options required.

The guiding principles of the OS project are:

- *The standards should be independent of programming language, operating system, and hardware.*

- *The standards should be open and available for everyone in the OR community to use free of charge.*
- *Optimization should be as easy as hooking up to the network.*

1.1. Optimization Services: Beyond NEOS

The OS project was initially motivated by the NSF grant, *Next-Generation Servers for Optimization as an Internet Resource.* The purpose of the grant was to design the technology for the next generation Network Enabled Optimization System (NEOS). See `www-neos.mcs.anl.gov`, Czyzyk et. al. (1998) and Moré et al. (2004) for more information on NEOS. (See also Czyzyk et al. (2000) and Fourer and Goux (2001) for earlier work on optimization using the Internet.) However, the OS project has become much more expansive than the NEOS project. While the OS project retains the goals of NEOS, NEOS is constrained by the following limitations, which the OS project goes beyond.

- There is no NEOS standard for communicating general optimization problem instances. The nearly 50 solvers in the NEOS lineup require instance inputs of about a dozen different kinds, including MPS and LP formats for linear and integer programming, SMPS extensions to the MPS format for stochastic programming, formats such as SDPA specific to semidefinite programming, DIMACS min-cost flow and other formats for network linear programming, and proprietary formats used by two modeling language processors. Other solvers recognize input programmed as functions in various languages including Fortran, C, C++, and Matlab.
- There is no NEOS standard for communicating the solver result of an optimization.
- There is no NEOS standard for communicating options to solvers.
- There is no NEOS common application programming interface (API) for solvers or modeling languages.
- NEOS is a centralized system with all solvers connected to a central server and all optimization requests must go through this server. There is no decentralized registry service for solvers to publish and register their services, nor is there any way for clients to discover these services. In short, NEOS is not a Service Oriented Architecture (see Section 2).

• There is no NEOS standard for communication between the components that are part of the system. See Section 4 for Optimization Services communication protocols.

• NEOS is not based upon the increasingly popular technology of Web Services. The use of Web Services overcomes some of the issues mentioned above.

How the Optimization Services project addresses all of the problems mentioned above is detailed in the rest of the paper.

1.2. Paper Outline

Figure 1 presents a summary of the OS protocols discussed in this paper. There are other OS protocols, but the ones in Figure 1 convey the major aspects of the OS framework without getting bogged down in too much detail. A description of all the OS protocols is found in Ma (2005) which is available online at `https://www.coin-or.org/OS/publications/Thesis2005.pdf`.

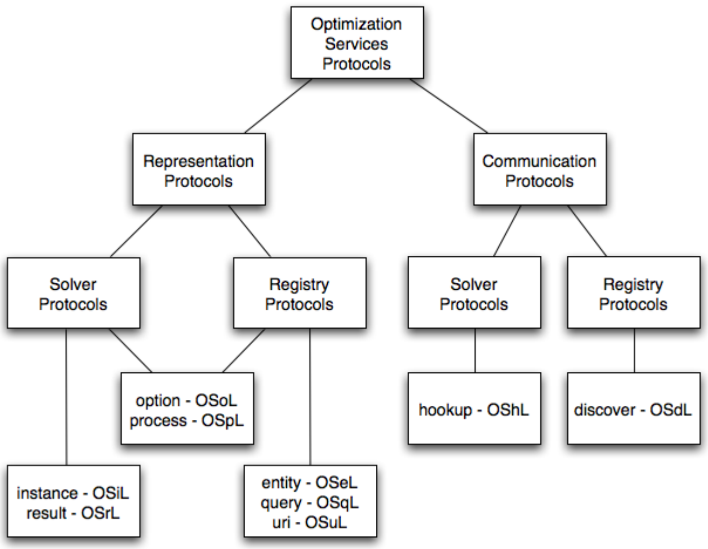


Figure 1 A Summary of OS Protocols.

The OS protocols are classified as either *representation protocols* or *communication protocols*. The communication protocols are "higher level" protocols than the representation protocols. Loosely speaking, the communication protocols specify what data are exchanged between client and server and the representation protocols specify detailed information about the data format. For example, a communication protocol might specify that in order to solve an optimization problem a client must send to the solver server a problem instance and solver options. The representation protocol would specify detailed information about the explicit format of the model instance and solver options.

Communication and representation protocols may apply to either a client communicating with a server that performs an optimization service or a client communicating with a server that provides a registry service. By an optimization service server we mean loosely a server with a solver that performs optimization on an optimization instance, analyses a problem, performs only a preprocessing service, or perhaps does simulation but not true optimization. A registry service server is one that allows servers with an optimization service to register its service or allows a client to discover servers that perform optimization services.

In the next section we provide sufficient background information on XML, Service Oriented Architecture, and Web Services in order to make the protocols outlined in Figure 1 understandable. In Section 3 we describe the key representation protocols. We describe OSiL (Optimization Services instance Language) for representing problem instances, OSrL (Optimization Services result Language) for representing optimization results, and OSoL (Optimization Services option Language) for representing options to solvers or registry services. We also describe OSqL (Optimization Services query Language) which is used to query a registry about which optimization services are available, and OSeL (Optimization Services entity Language) which is used by an optimization service to register with the registry server.

In Section 4 we describe OS communication protocols. The communication between a client and a server with an optimization service is specified by the OShL (Optimization Services hookup Language) protocol. The communication between a client and a server with a registry service is specified by the OSdL (Optimization Services discovery Language) protocol. We detail the key aspects of both of these communication protocols in this section.

The OS framework is just that – a framework. It does not specify implementation details, programming languages, etc. However, in order to provide a reference implementation, or proof a concept, many of the protocols described in this paper are implemented in a set of open source libraries. These libraries, along with the associated source code, have been donated to COIN-OR (COmputational INfrastructure for Operations Research) and constitute the COIN-OR OS project. In Section 5 we give a brief description of the COIN-OR OS project. The last section of the paper, Section 6, provides conclusions and a brief outline of future work. Appendices with examples are available in electronic format online.

2. Background Material

In this section we give a brief introduction to XML, Service Oriented Architecture, and Web Services. These concepts are used throughout the rest of the paper.

2.1. XML

All of the OS protocols are expressed in XML (Extensible Markup Language). The choice of XML for the OS protocols was obvious for the following reasons: 1) all of the Web Services protocols are expressed in XML; 2) the language of the Web, HTML (Hypertext Markup Language), is being replaced by the XML version, XHTML; and 3) XML is becoming the *lingua franca* of data. XML, in short, is simply the best way to store text data. The XML standard is controlled by the W3C (World Wide Web Consortium); see W3C (2007). For an overview of XML technologies see Skonnard and Gudgin (2002).

In order to understand the OS protocols we provide a brief introduction to XML. XML is a markup language. An XML string or file contains both *data* and *markup* that describe the data. The presence of markup makes it easy to parse the data. The markup used to describe the data consists of *tags,* that are illustrated below. These tags must be organized according to certain general principles, but they are quite flexible in their meaning. The following problem instance (which is a modification of an example of Rosenbrock (1960)) is used to illustrate XML and other concepts throughout the paper.

$$\text{Minimize} \quad (1-x_0)^2 + 100(x_1 - x_0^2)^2 + 9x_1 \tag{1}$$

$$\text{Subject to} \quad x_0 + 10.5x_0^2 + 11.7x_1^2 + 3x_0x_1 \le 25 \tag{2}$$

$$\ln(x_0 x_1) + 7.5x_0 + 5.25x_1 \geq 10 \tag{3}$$

$$x_0, x_1 \geq 0 \tag{4}$$

In the model defined by (1)–(4) there are two continuous variables, x_0 and x_1, each with a lower bound of 0. Figure 2 shows how this variable information could be stored as XML data. As with all XML data, there are both markup and data in this example. In this case there are two types of markup, *elements* (or *tags*) and *attributes* that describe the elements. Specifically, there is a `<variables>` element and two `<var>` elements. The `<variables>` element is used to mark the start and end of the `<var>` elements. The two `<var>` elements correspond to x_0 and x_1, and each `<var>` element has attributes `lb`, `name`, and `type` that describe properties of a decision variable: its lower bound `lb`, `name`, and domain `type`.

```
<variables numberOfVariables="2">
    <var lb="0" name="x0" type="C"/>
    <var lb="0" name="x1" type="C"/>
</variables>
```

Figure 2 The `<variables>` Element Example (1)–(4).

The actual values of the attributes, such as `"0"` (zero) for `lb` and `"C"` (denoting a continuous domain) for `type`, are the data in the file. An attribute may also assume a default value when it does not appear. For example, the `<var>` element has a `ub` attribute that is absent in Figure 2 and that consequently takes the default value `"INF"` (denoting ∞).

In the XML representation of the variable data illustrated in Figure 2, the text markers surrounding each tag (`<` and `>`), as well as other elements of the XML syntax, serve a very important purpose: they make XML instances very easy to parse and to validate. Numerous parsers, both open source and proprietary, are available for parsing an XML document.

2.2. Service Oriented Architectures

A standard distributed computing architecture appears in Figure 3. In this architecture, there is a central server that acts a "middleman" between all of the clients and all of the other servers. All client requests must go through this central server. The NEOS architecture is a good illustration of the central server model. All optimization instances (which could be very large) and solutions must pass through the central server which then schedules a solver server to optimize the model and pass the result back to the client through the central server.

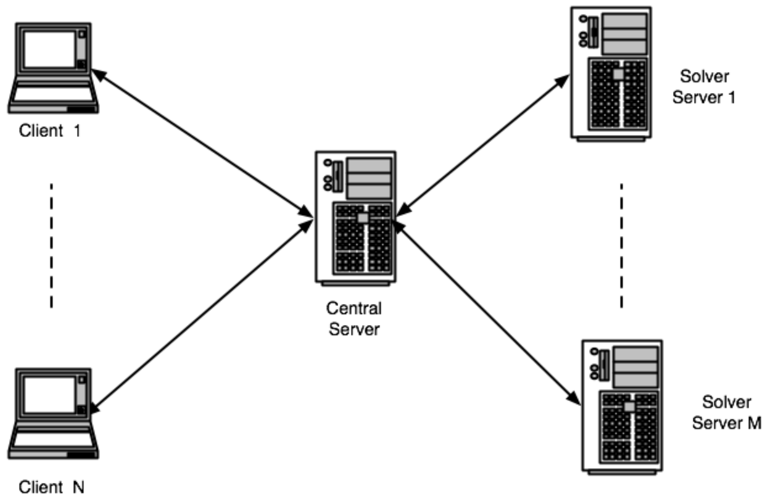


Figure 3 Centralized Distributed Computing Architecture.

The central server model does not scale up well. Indeed, the Internet works well because it has a decentralized architecture; there is no such thing as a "central web server" through which all requests pass. Rather, there are directory services such as Google and Yahoo where a client can look up the resource of interest and then contact that resource directly rather than going through a central server. Following the Internet model, IT departments are increasingly turning to (SOA) service oriented architectures when building their infrastructure. See Figure 4.

In the SOA paradigm, a service provider (in this case an optimization service such as a solver or analyzer) registers with a registry/discovery service. In a sense, this registry discovery/service does act as central server, but it functions as a lightweight registry that only maintains information about available service providers. It does not interact directly with the service provider to provide the service to the client. The client or service consumer then "discovers" the service that is described in the registry. Then, rather than interact directly with the registry/discovery server to consume the service from the service provider, the service consumer contacts the service provider and they work in a "peer-to-peer" fashion.

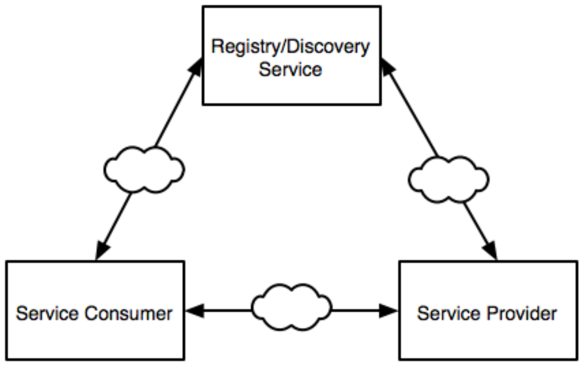


Figure 4 Services Oriented Architecture (SAO) Paradigm.

In Figure 5 we show the SOA version of the distributed optimization system first illustrated in Figure 3. The key contrast between the two architectures is that in the SOA version illustrated in Figure 5 the clients and solvers are exchanging optimization instances and results directly in a peer-to-peer mode, thus enabling scaling of the system.

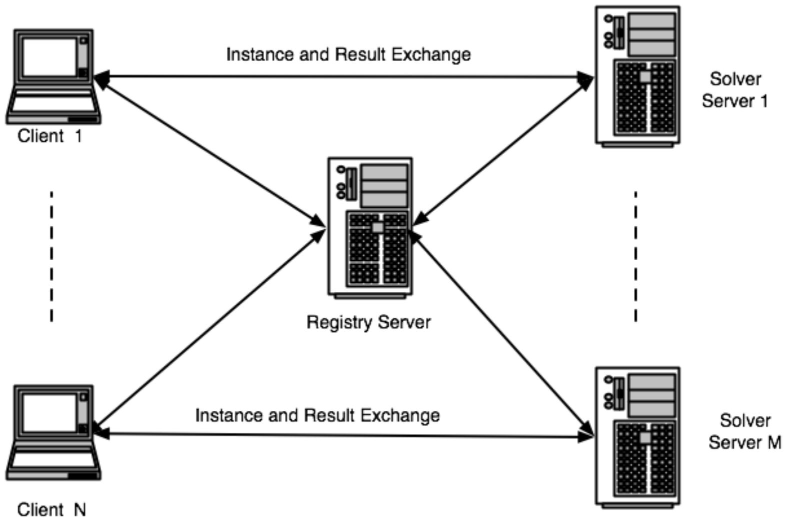


Figure 5 Centralized Distributed Computing Architecture.

SOA is a philosophy of how to build a decentralized architecture. It is not a set of actual protocols or standards. In Section 2.3 we describe the Web Services protocols for actually implementing an SOA.

2.3. Web Services

Web Services consists of three XML-based protocols. They are SOAP (Simple Object Access Protocol), WSDL (Web Services Discovery Language), and UDDI (Universal Description, Discovery, and Integration).

In an SOA architecture, service consumers make *requests* to service providers and get *responses* from the provider. SOAP is an XML-based protocol that specifies how information should be encoded in the request and response messages. This information is then sent over the network using an application layer protocol such as HTTP (Hyper Text Transfer Protocol), FTP (File Transfer Protocol), or SMTP (Simple Mail Transfer Protocol). In Figure 6 we illustrate SOAP over HTTP.

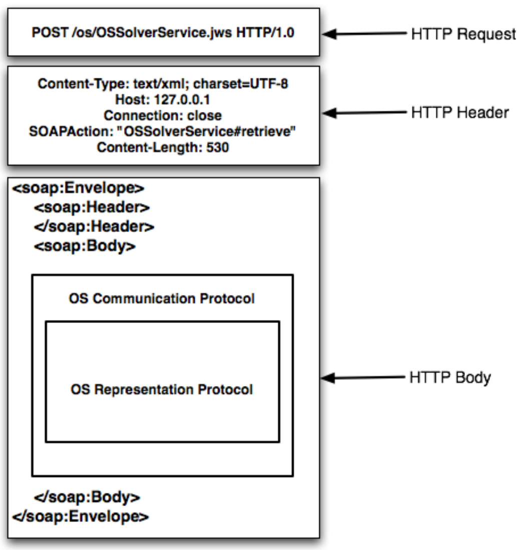


Figure 6 OS Protocols Inside a SOAP Envelope Inside an HTTP Body.

In Figure 6 OS representation protocols (see Section 3) are packaged inside OS communication protocols (see Section 4) that are in turn packaged inside a SOAP envelope that constitutes the body of the HTTP message. For example, a client may send an optimization instance in the OSiL protocol, and solver options in the OSoL protocol using an OShL communication protocol that instructs a solver to optimize the problem. All of these protocols would be in a SOAP envelope in an HTTP body sent over the Internet using the HTTP protocol to the solver service.

The OS communication protocols are expressed using WSDL. WSDL is a protocol for expressing, in XML format, the methods (functions) and arguments provided by a Web service. WSDL is used by the provider of a service to tell the consumer of service how to consume the service. This will become more clear in Section (4).

Finally UDDI is an XML-based protocol describing how providers can join the registry and how consumers can query the registry. We discuss registry protocols in Section 4.2.

The next two sections of the paper describe the "plumbing" of the OS Framework. Before proceeding, it is important to emphasize that the typical user of a modeling system based on OS will never have to be aware of what is presented in Sections 3 and 4. For example, a user of the AMPL modeling language would build a model as usual and then issue a command inside AMPL such as

```
option ipopt_options "service http://gsbkip.chicagogsb.edu/os/OSSolverService.jws";
```

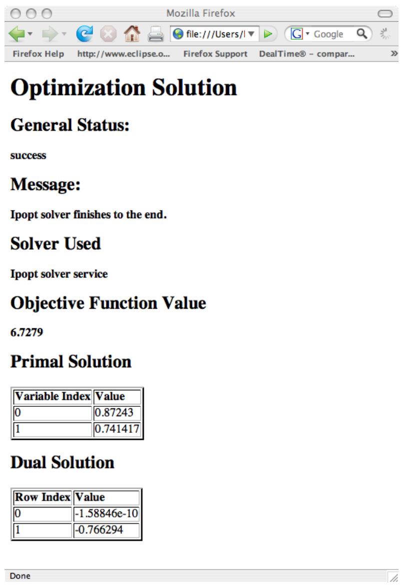
and this will have the effect of using the solver Ipopt on a machine with the address `gsbkip.chicagogsb.edu/os/OSSolverService.jws`. Then, using tools provided in the OS COIN-OR project, the optimization solution can be displayed for the user in a browser window as shown in Figure 7.

3. Optimization Services Representation Protocols

In the OS framework there are representation protocols for communication with solver servers and registry servers. We first discuss the representation protocols for communication with a solver server.

3.1. Representation Protocols for Optimization Servers

Currently there are no standards for communicating general optimization model instances, the results of model optimizations, or options to an optimization solver. In a tightly coupled environment where the optimization model instance generator and optimization solver are coupled



Optimization Solution

General Status:

success

Message:

Ipopt solver finishes to the end.

Solver Used

Ipopt solver service

Objective Function Value

6.7279

Primal Solution

Variable Index	Value
0	0.87243
1	0.741417

Dual Solution

Row Index	Value
0	-1.58846e-10
1	-0.766294

Figure 7 The optimization result displayed in a browser.

together as a single piece of software this is not a problem. A good example of this is LINGO (see `www.lindo.com`) which is both a modeling language and a solver. Similarly, Cplex (`www.ilog.com`) is a solver with model generation capability bundled as a single product. However, beginning with the development of popular standalone modeling languages such as GAMMS and AMPL in the early 1980s, it became increasingly common to have one piece of software for generating a model instance and a separate one for optimizing that instance. Then NEOS appeared in the mid-1990s completely breaking the link between model generator and solver, allowing a model to be developed on one machine and then sent over the network to a solver on another machine.

The downside of these developments is that we now have a huge proliferation of modeling languages and solvers. NEOS alone has nearly 50 solvers that require instance inputs of about a dozen different kinds. This is now a significant problem for software developers in the optimization community. If there are M modeling languages and N solvers, then $M \times N$ "drivers" are required for complete interoperability. However, if there are standards for representing model instances, model results, and solver options, then only $M + N$ drivers are required for interoperability. In the following sections we describe the OS standards for representation of instances, solutions, and solver options.

3.1.1. OSiL: Optimization Services Instance Language OSiL is an XML-based language for representing instances of large-scale optimization problems including linear programs, mixed-integer programs, quadratic programs, and very general nonlinear programs. We illustrate the major features of OSiL using the optimization model given in Equations (1)-(4) in Section 2.1.

There are two continuous variables, x_0 and x_1, in this instance, each with a lower bound of 0. Back in Figure 2 we showed how we represent this information in XML using both data and markup (tags or elements). We chose the name `<var>` for each element that represents a variable. Rather than using the `<var>` abbreviation we could have named each variable element `<variable>` instead. Clearly, there are countless ways to represent an optimization instance in XML. However, when parsing an XML file there can't be any ambiguity, so in order to be useful for communication between solvers and modeling languages, the markup in the instance files must conform to a standard naming convention.

A common way to impose a standard on the naming and structure of an XML file is to use the *W3C XML Schema* language. All of the OS representation protocols are specified by a W3C XML Schema. A *schema* specifies the elements and attributes that define a specific XML vocabulary. The W3C XML Schema is thus a schema for schemas; it specifies the elements and attributes for a schema that in turn specifies elements and attributes for an XML vocabulary such as OSiL. An XML file that conforms to a schema is called *valid* for that schema. Indeed, when we talk about an "Optimization Services instance Language," we are really talking about the OSiL schema. The schema defines the language or vocabulary. All of the OS schemas are available at `http://www.optimizationservices.org/`. Readers who wish to view and/or work with schemas may find software such as XML Spy or Oxygen very useful. These packages present schemas in a nice graphical format.

By analogy to object-oriented programming, a schema is akin to a header file in C++ that defines the members and methods in a class. Just as a class in C++ very explicitly describes member and method names and properties, a schema explicitly describes element and attribute names and properties.

Recall that Figure 2 is the XML representation of the variables for the Rosenbrock model shown in Equations (1)–(4). To show how Figure 2 conforms to the OSiL schema, Figures 8 and 9 are needed. Figures 8 and 9 are the part of the OSiL W3C XML Schema that defines the standard for what every variables section of an instance in OSiL should look like, of which Figure 2 is an example. In particular, Figure 8 is the schema specification of the element `<variables>` of Figure 2, and Figure 9 is the schema specification of the element `<var>` of Figure 2.

In Figure 8, a complexType named Variables is defined. Just as many programming languages such as C++ and Java allow the user to define their own data types that extend standard data types such as integer, double, and string, the W3C XML Schema standard also permits user-defined data types called a complexType. The complexType is used to specify the elements and attributes that are allowed to appear in a valid XML instance file such as the one shown in Figure 2. The Variables complexType of Figure 8 is made up of a sequence of elements named `<var>`. The `<var>`

```
<xs:complexType name="Variables">
    <xs:sequence>
        <xs:element name="var" type="Variable" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="numberOfVariables"
            type="xs:positiveInteger" use="required"/>
</xs:complexType>
```

Figure 8 The `Variables` complexType in the OSiL schema.

```
<xs:complexType name="Variable">
    <xs:attribute name="name" type="xs:string" use="optional"/>
    <xs:attribute name="init" type="xs:string" use="optional"/>
    <xs:attribute name="type" use="optional" default="C">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="C"/>
                <xs:enumeration value="B"/>
                <xs:enumeration value="I"/>
                <xs:enumeration value="S"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="lb" type="xs:double" use="optional" default="0"/>
    <xs:attribute name="ub" type="xs:double" use="optional" default="INF"/>
</xs:complexType>
```

Figure 9 The `Variable` complexType in the OSiL schema.

element is a user-defined complexType named Variable. The Variable complexType is defined in Figure 9, which is part of the OSiL W3C XML Schema. Thus the `<var>` element as it appears in Figure 2 conforms to this schema.

The Variables complexType in Figure 8 also has an attribute named `numberOfVariables` that specifies the number of `<var>` elements in the XML instance file. The `numberOfVariables` attribute is the standard type positiveInteger. Note in Figure 2, `numberOfVariables` is defined to be 2, which is a positiveInteger. Also in Figure 2, note the `<var>` elements nested inside of the `<variables>` element. In our convention, the elements that appear in an actual XML instance file have element names that are all lower case, and the corresponding complexType in the schema begins with an upper case letter. The complexType appears only in the schema and not in the XML instance file; the XML instance file is made up of elements that are instances of the complexTypes defined in the schema. Thus, the `<variables>` element of Figure 2 contains a sequence of `<var>` elements that are instances of complexType Variable defined in Figure 9, and so `<variables>` conforms to its definition in the OSiL W3C XML Schema.

There is a complexType for each major part of a general optimization problem. These parts are `<variables>`, `<objectives>`, `<constraints>`, `<linearConstraintCoefficients>`, `<quadraticCoefficients>`, and `<nonlinearExpressions>`. We have already discussed the `<variables>` section. The `<objectives>` and `<constraints>` are very similar to the the `<variables>` element. We briefly discuss `<linearConstraintCoefficients>`, `<quadraticCoefficients>`, and `<nonlinearExpressions>` next.

Many large-scale problems are mostly linear and it makes sense to store the linear part of the problem using sparse techniques. In OSiL, the linear part of the problem is stored in the `<linearConstraintCoefficients>` element, which stores the coefficient matrix using the standard three-array sparse matrix storage scheme. The `<linearConstraintCoefficients>` element has child element `<value>` that corresponds to an array of nonzero coefficients, a child element `<rowIdx>` or `<colIdx>` for a corresponding array of row indices or column indices, and `<start>` for an array that indicates where each row or column begins in the previous two arrays.

```
<ln>
     <times>
          <variable coef="1.0" idx="0"/>
          <variable coef="1.0" idx="1"/>
     </times>
</ln>
```

Figure 10 The OSiL Representation of $ln(x_0x_1)$.

There are specialized codes for quadratic programming problems so in OSiL it is possible to store quadratic terms separately from the other more general nonlinear terms. The quadratic part of the problem is in the `<quadraticCoefficients>` element. Each quadratic term in the problem is stored as a triple: the index for each of the two variables and the coefficient on the term. The row index is also recorded.

Finally, general nonlinear expressions are easily represented in OSiL. The key to representing general nonlinear expressions in XML is to use the construct of a `substitutionGroup`. A detailed explanation of how the `substitutionGroup` works is beyond the scope of this paper. See Fourer, Ma, and Martin (2007) for a thorough explanation. However, the basic construct is an `OSnLNode`; every nonlinear operator, such as `OSnLNodeLn`, which corresponds to the natural logarithm, is of type `OSnLNode`. The benefits of this approach are also in Fourer, Ma, and Martin (2007). To see what the actual XML might look like, in Figure 10 we give the OSiL representation of the term $ln(x_0x_1)$ that appears in Equation 3. See Fourer, Ma, and Martin (2007) for a thorough discussion of the OSiL protocol.

3.1.2. OSrL: Optimization Services Representation Language OSrL is an XML-based protocol for representing the solution of large-scale optimization problems including linear programs, mixed-integer programs, quadratic programs, and very general nonlinear programs. As with OSiL, OSrL is defined by an XML schema. In developing the OSrL schema our design goal was to maximize flexibility in reporting optimization results but keep the design simple.

With OSiL the objective is to be as encompassing and complete as possible. We want to represent all interesting optimization instances. With OSrL we take a minimalist approach. A linear program is a well-defined entity but the solution of a linear program is not. We cannot have a linear program without constraints or variables; however, we can have a linear programming solution without reduced cost or allowable increase/decrease information. Different linear or nonlinear optimization solver codes may present their results in different formats; some may include more detail than others. For example a linear programming code may or may not provide optimal value function data for right-hand-side and objective-function coefficients. The level of solution detail is up to the solver developer and would be difficult to standardize. The objective of the OSrL schema is to the allow the solver developer to report as much or little detail as desired.

We illustrate this design philosophy with the OSrL `<variables>` element. This element has two child elements. The first child element is `<values>`. At the minimum, the solver developer will probably want to report the values of the variables in a feasible or optimal solution. Hence the presence of the `<values>` element with the `idx` attribute to index the variables. However, after the values of the decision variable, it is not clear what solution information associated with variables the solver developer will wish to report. Hence, we have an `<other>` element with attributes `name` and `description`. The `<variables>` can have none or an unlimited number of `<other>` children. This allows the solver developer complete flexibility in reporting.

```
<variables>
    <values>
    <var idx="0">539.984</var>
    <var idx="1">252.011</var>
    </values>
    <other name="reduced costs"
         description="the variable reduced costs">
    <var idx="0">0</var>
    <var idx="1">0</var>
    </other>
 </variables>
```

Figure 11 An OSrL `<variables>` **Element.**

In Figure 11 we illustrate a `<variables>` element. For each variable the solution value is reported with the associated variable index. The `<other>` element is used to report reduced cost information for each variable. There are similar constructs for, among others, `<constraints>`

and `<objectives>`. Readers wishing to see more detail can view the OSrL schema at `www.optimizationservices.org`.

3.1.3. OSoL: Optimization Services Option Language OSoL is an XML-based language for representing solver options for optimization problems including linear programs, mixed-integer programs, quadratic programs, and very general nonlinear programs. As with OSiL, OSoL is defined by an XML schema. The philosophy of the OSoL design is identical to the OSrL philosophy – maximize flexibility in reporting solver options but keep the design simple. Solver options vary greatly from code to code and there is no standardized set of solver options. Like OSrL, OSoL provides an `<other>` element that solver developers can make extensive use of for their specialized options.

```
<osol xmlns="os.optimizationservices.org">
    <general>
        <contact transportType="smtp">
            kipp.martin@chicagogsb.edu
        </contact>
        <instanceLocation locationType="http">
           http://www.coin-or.org/OS/rosenbrockmod.osil
        </instanceLocation>
    </general>
    <optimization>
        <other name="solverAlg">barrier</other>
        <other name="tolerance">0.001</other>
    </optimization>
</osol>
```

Figure 12 An Example of OSoL.

Figure 12 is a good illustration of the flexibility of OSoL. The `<general>` element has an optional child `<contact>` element. The `<contact>` element attribute `transportType` has value `smtp` which tells the solve to email the results of the optimization to the address contained in `<contact>` element. There is also an `<instanceLocation>` element with `locationType` equal to `http`. This tells the solver the location of the model instance. This allows a client machine to call a remote solver server and tell the solver server where to get the model instance. The client machine does not need to actually send the model instance, only the location.

Within the `<optimization>` element are two `<other>` tags. The first `<other>` tag conveys to the solver to use a barrier solution algorithm. The second `<other>` tag conveys a tolerance to the solver. If a solver cannot interpret the value of an attribute for an `<other>` tag it simply ignores it and uses only the options it understands. The OSoL protocol is also used to pass options to a registry server so it is both an optimization and registry representation protocol. Readers wishing to see more detail can view the OSoL schema at `www.optimizationservices.org`.

3.1.4. In-Memory Representation When a model instance is sent over the network as an OSiL file to a solver service as in Figure 5, the solver service must read this OSiL file, extract all the information about the optimization instance, and put this information into its own internal data structures. To aid in this process, we have written open-source C++ libraries (see Section 5) that provide an interface to the model instance. At the heart of this interface is the `OSInstance` class. The OSInstance class has a set of methods (functions) that allow the user of the OS library to:

• Get information about the model instance through a set of `get()` methods. For example, the user can get information on the number of variables, number of constraints, lower and upper bounds on the constraints, all of the coefficients for the linear terms, etc; in short, any information about the instance that an optimizer would require.

• Modify the existing instance, or even create a new model instance from scratch through a set of `set()` methods.

• Provide information to solvers that require function evaluations, Jacobian and Hessian sparsity patterns, function gradient evaluations, and Hessian evaluations through a set of `calculate()` methods. The gradient and Hessian calculations are made using algorithmic differentiation (the OS libraries are linked to the COIN-OR project CppAD for algorithmic differentiation. See Bell (2007)).

Similarly, given any OSrL file or string representing a problem solution, there is a corresponding in-memory `OSResult` class that can be used to extract information about the model solution.

3.2. Representation Protocols for Registry Servers

When an optimization service registers with a registry service it must provide detailed information about the services it can solve. For example, is it only a linear programming solver, or does it allow integer variables. If it is a nonlinear solver, does it allow only quadratic terms or perhaps more general nonlinear terms. In the OS framework, the information the optimization service must provide the registry service is specified using the OSeL (Optimization Services entity Language). This is a representation protocol specified by a schema just as the representation protocols for optimization servers are specified by a schema. A key element specified by the OSeL schema is `<optimizationType>`. This element contains numerous children such as `<contraintType>`, `<variableType>`, etc. that spell out the specific kinds of optimization problems that the solver can solve.

If a client is to communicate with an optimization server with required type of solver in peer-to-peer fashion it must have the address (URL) of the solver server. In order to find which solvers can solve the required optimization type, the client will query the registry server. To query the database on the registry server, clients use the Optimization Services query Language (OSqL) which is a schema designed for a query protocol. The OSqL schema specifies, for example, an `<optimizationType>` element matching that of the OSeL schema. This allows for the symmetric registration and querying of optimization problem types.

It is important to note that the OS Framework *does not* specify how the registry service should parse the OSqL query and use the information to query the registry database. One possibility is to parse the OSqL query and build a query in XQuery that is executed against the registry database if the registry database is in XML format. XQuery is a standard specified by the W3C and is designed to query XML databases. XQurey is to an XML data base what SQL is to a relational database. However, OS protocols do not specify formats or standards for the registry database. It is certainly possible to implement the OS framework by using a standard relational database for the registry.

How the OSqL query is communicated to the OS registry is specified in the Optimization Services discover Language (OSdL), a WSDL document. The clients get the location information about optimization service solvers from the registry as a sequence of URIs (or URLs). This syntax is specified in the Optimization Services uri Language (OSuL). The OSdL protocol is discussed in Section 4.2.

4. Optimization Services Communication

We have described OS protocols for representing model instances, optimization results, and solver options. In a distributed computing environment these representations must be *communicated* between the service consumers and service providers. In this section we describe the OS communication protocols. First we describe communication protocols for communicating with solver servers.

4.1. Communication Protocols for Optimization Servers

Refer back to Figure 4 which illustrates the SOA paradigm. For there to be effective communication between a consumer and a provider, service consumers must tell service providers exactly "what to do." Similarly, service providers must tell service consumers what "they can do." In a Web Services implementation of a service oriented architecture, a service provider communicates to consumers its *capabilities* using WSDL (Web Services Discovery Language). By "capabilities" we mean a set of functions or methods that the provider can perform. A WSDL document is written in XML and provides a listing of these methods along with their arguments (inputs) and outputs.

In the OS project the key communication protocol is OShL (Optimization Services hookup Language) and it describes the methods to be used in communication between solvers and clients. For example, when communicating with a service provider that provides an optimization service, a natural method is `solve()`. The service consumer will request a `solve()` service from the solver service provider. In Figure 13 is the WSDL (written in XML) that defines the `solve()` method. The method takes a `solveRequest` and responds with a `solveResponse`.

Figure 14 provides the details of the `solveRequest` and `solveResponse`. A `solveRequest` requires two string arguments: the first argument `osil` is the model instance in OSiL format and

```
<operation name="solve" parameterOrder="osil osol">
     <input name="solveRequest" message="os:solveRequest"/>
     <output name="solveResponse" message="os:solveResponse"/>
</operation>
```

Figure 13 The `solve()` Method.

the second argument `osol` is the solver options in OSrL format. The `solveResponse` is `osrl` which is the solution in an OSrL string.

```
<message name="solveRequest">
     <part name="osil" type="xsd:string"/>
     <part name="osol" type="xsd:string"/>
</message>
<message name="solveResponse">
     <part name="osrl" type="xsd:string"/>
</message>
```

Figure 14 The `solveRequest` and `solveResponse`.

Below is a summary of the inputs, outputs, and purpose of the six methods that constitute `OShL`. See also Figure 15.

• `solve(osil, osol):` takes as input a string with the instance in OSiL format and a string with the solver options in OSoL format and returns a string with the solver solution in OSrL format. The solve method should be used for *synchronous* communication with the server. The `solve` method waits for the solution of the model.

• `send(osil, osol):` takes as input a string with the instance in OSiL format and a string with the solver options in OSoL format and returns a boolean, true if the problem was successfully submitted, false otherwise. This method should be used for *asynchronous* communication with the server. When using this method the `osol` string should have a `JobID` in the `<jobID>` element. This method can be used with `knock()` to see if a job is ready and with `retrieve()` to get the results back.

• `getJobID(osol):` takes as input a string with the solver options in OSoL format (in this case, the string may be empty because no options are required to get the JobID) and returns a string which is the unique JobID generated by the solver service. This is used to maintain session and state on a distributed system. The JobID returned can be used as input in the `osol` string for the `send()` method.

• `knock(ospl, osol):` takes as input a string in OSpL format and a string with the solver options in OSoL format and returns process and job status information from the remote server in OSpL format. This method can be used to see if a job is complete, and if so, `retrieve()` can be used to get the result. OSpL is Optimization Services Process Language. It serves as both an optimization and a registry protocol.

• `retrieve(osol):` takes as input a string with the solver options in OSoL format and returns a string with the solver solution in OSrL format. The `osol` string should have a `JobID` in the `<jobID>` element.

• `kill(osol):` takes as input a string with the solver options in OSoL format and returns process and job status information from the remote server in OSpL format. This method can be used to terminate long running jobs or jobs for which there was input error.

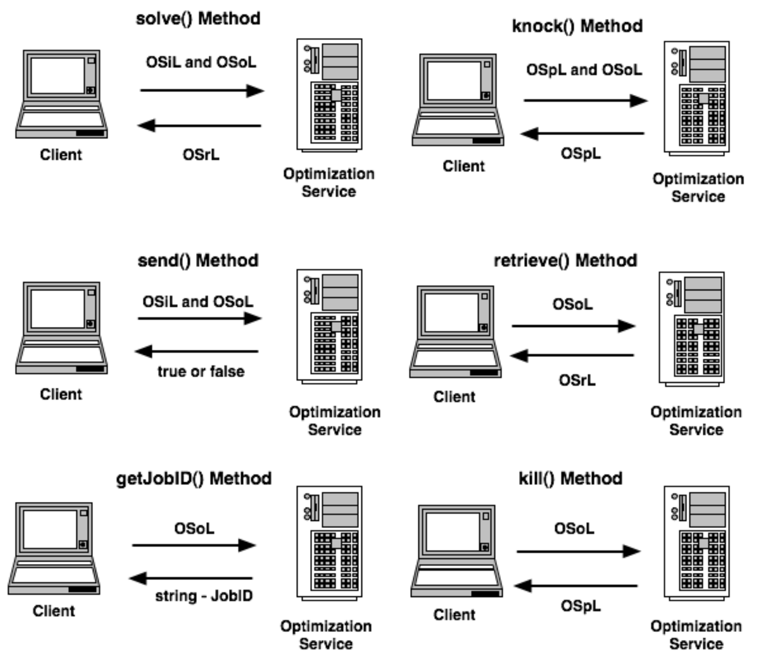


Figure 15 **The OS Hookup (OShL) Communication Methods.**

Refer to the Electronic Appendix for a detailed example illustrating the OShL protocols.

Recall the Web Services and SOAP discussion back in Section 2.3. Services providers and service consumers communicate with each other through SOAP. See Figure 6. If, for example, a service consumer wished to retrieve the result of an optimization the consumer could send a SOAP envelope in the body of an HTTP message to the solver server. The SOAP envelop would contain a `<retrieve>` tag. The `retrieve()` method has a single argument `osol`. Thus the `<retrieve>` tag has a child tag `<osol>`. The `<osol>` tag then contains the actual solver options in XML format conforming to the OSoL schema. Therefore we have a string in the OSoL protocol packed inside a tag in the OShL protocol in a SOAP envelope in an HTTP body!

An additional benefit of using WSDL is that when a user implements an OS Web service on the server side, they can use OShL as a reference and take advantage of software tools that will automatically generate much of the needed server side code.

4.2. Communication Protocols for Registry Servers

In the OS framework, clients must be able to *discover* the location of optimization solvers in order to initiate direct peer-to-peer communication with them. Second, optimization solvers must be able to *register* their service. The Optimization Services discover Language (OSdL) specifies the protocol for communicating with the registry server in order to both register and discover optimization services. Like OShL, OSdL is specified using WSDL. The two key methods described by the OSdL WSDL are `find` and `register`. See Figure 16.

The `find` method is used to discover an optimization service. This method takes two arguments. The first argument is a string that follows the OSqL protocol. This string contains the query commands for finding the necessary information about an optimization solver; for example, can it solve nonlinear optimization problems. The second argument is a string in the OSoL format. One might use the options to specify, for example, a limit on the number of results returned. The method returns a string in the uri (OSuL) protocol which will have the URL for solvers capable of performing this task. The `register` method is used by the optimization service to register itself with the registry service. The information about the optimization service is passed to the registry

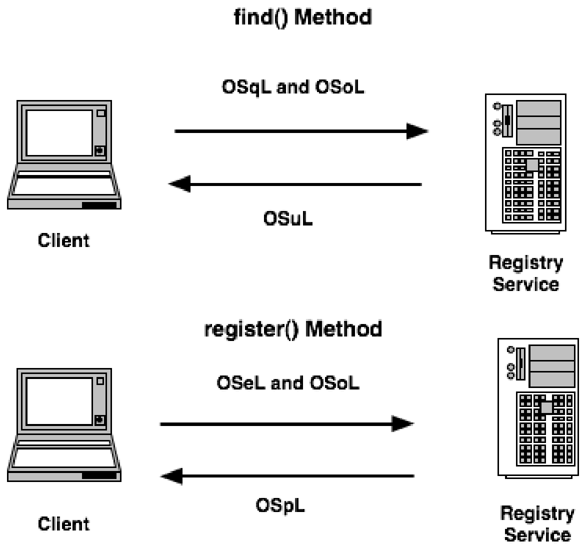


Figure 16 **The OS Discovery (OSdL) Communication Methods.**

service using the entity (OSeL) protocol. The OSeL, OSqL, and OSuL representation protocols were described in Section 3.1.

5. The COIN-OR Open Source OS Project

In order to provide a reference implementation and test the OS framework, we have implemented many of the protocols described in this paper in C++ and Java libraries. This code is the basis of an open-source project that is part of the COIN-OR (COmputational INfrastructure for Operations Reseach). See `projects.coin-or.org/OS`. The COIN-OR OS project provides the following.

1. A library of classes for reading and writing files in the OSiL, OSoL, and OSrL format.

2. A robust solver and modeling language interface (API) for linear and nonlinear optimization problems. Corresponding to the OSiL problem instance representation there is an in-memory object, `OSInstance`, along with a set of `get()`, `set()`, and `calculate()` methods for accessing and creating problem instances. This is a very general API for linear, integer, and nonlinear programs and was described in Section 3.1.4.

3. A command line executable `OSSolverService` for reading problem instances (OSiL format, AMPL nl format, MPS format) and calling a solver either locally or on a remote server. The `OSSolverService` implements the `solve()`, `send()`, `getJobID()`, `knock()`, `retrieve()`, and

`kill()` methods described in Section 4. A detailed illustration of using the `OSSovlerService` in conjunction with the OShL methods is given in the Electronic Appendix.

4. Utilities that convert AMPL nl files and MPS files into the OSiL XML format.

5. A library that can be used to create Web Services SOAP packages with OSiL instances and contact a server for solution.

6. An executable program `OSAmplClient` that works with the AMPL modeling language. The `OSAmplClient` appears as a "solver" to AMPL and, based on options given in AMPL, contacts solvers either remotely or locally to solve instances created in AMPL in the AMPL modeling language.

7. Server software that works with Apache Tomcat and Apache Axis. Included in the COIN-OR project is a Web Services implementation `OSSolverService.jws`. This acts as middleware between the remote client that submits the instance and the solver on the server that optimizes the instance and returns the result. This Web service implements the `solve()`, `send()`, `getJobID()`, `knock()`, `retrieve()`, and `kill()` methods on the server end.

8. A lightweight version of the project, `OSCommon` for modeling language and solver developers that want to use OS API, readers, and writers, without the overhead of other COIN-OR projects or any third party software.

See the Electronic Appendix, Section EC.1 for an illustration of using the COIN-OR software to call a remote server.

6. Conclusions and Future Work

We have provided a framework for using optimization as service. Future work is directed into two areas. First, we are working to extend the libraries to new classes of optimization problems. The COIN-OR libraries currently support linear, quadratic, and general nonlinear programming and continuous, binary, and general integer variables. We are currently in the process of extending these libraries to a broader variety of optimization problems. In particular, the authors are currently working on extensions for semidefinite and cone programming, robust optimization, disjunctive programming, constraint programming, and stochastic programming (see Fourer et al. (2007)).

Second, we are working to gain acceptance of the OS standard. The COIN-OR project is a start. We hope to use the COIN-OR project as springboard to get modeling language developers and solver developers to adopt the OS framework. In terms of solvers, the COIN-OR OS libraries currently support the commercial solvers Cplex, Knitro, and LINDO, in addition to the open-source COIN-OR solvers Cbc, Clp, DyLP, Ipopt, SYMPHONY, and Volume. The GNU Glpk solver is also supported by the OS libraries. The Mosek ApS optimization solver and the Frontline Systems Solver Platform SDK currently support OSiL for problem instance representation of mixed integer linear programs.

Support is also being developed for modeling languages. The COIN-OR project includes an executable `OSAmplClient` that can be called from inside the AMPL modeling language. The `OSAmplClient` will then take the AMPL instance, convert it to OSiL, and call a solver either locally or remotely. The optimization result in OSrL format is then translated back into a format understandable by AMPL for displaying results. See Dolan et al. (2002) for a related approach using Kestrel.

A similar feature is available for the GAMS modeling language through the COIN-OR project GAMSlinks `projects.coin-or.org/GAMSlinks` led by Stefan Vigerske. A prototype currently supports mixed integer linear programming. For example, if the user has downloaded the GAMS modeling language (see GAMS (2008) and Brooke et al. (1988)) a model is loaded much like in AMPL through the command `gamslib trnsport`. The model is solved through the OS libraries using the command `gams trnsport lp=os optfile=1`. This results in a model solution and display of results. An option can also be set in the GAMS option file to solve the problem remotely. We also plan to develop similar features for the LINGO modeling language. We illustrate this process form AMPL and GAMS in the Electronic Appendix Section EC.2 .

LogicBlox, an Atlanta company that develops online predictive and optimization software, is currently developing a product based on Optimization Services. This product will allow users to develop optimization models through a Web based GUI. The model instance is written as OSiL and then sent to a solver on a local or different machine and the underlying result in returned as

OSrL where it is then converted into a more user-friendly solution report. A browser is the only required software on the client machine. This is a true example of optimization as a service.

Acknowledgments

This paper has benefited greatly from careful reading and comments by Gail Honda.

References

Bell, B. 2007. CppAD Documentation, `http://www.coin-or.org/CppAD/Doc/cppad.xml`.

Brooke, A., D. Kendrick, A. Meeraus. 1988. *GAMS, A User's Guide*. Scientific Press, Redwood City, CA.

COIN. 2003. COIN LP. `http://www.coin-or.org/`.

Czyzyk, J. Owen, J.H. Wright, S.J. 1997. Optimization the Internet. *OR/MS Today.* **5** 48-51.

Czyzyk, J. Mesnier, J. Moré, J.J. 1998.. The NEOS server. *IEEE Journal on Computational Science and Engineering*, **5** 68-75.

Dolan, R.Fourer, Goux, J.-P. Munson, T.S., Kestrel: An Interface from Modeling Systems to the NEOS Server. Technical report, Department of Industrial Engineering and Management Sciences, Northwestern University.

Fourer, R. Gassman, G. Ma, J. Martin, K. 2007. An XML Based Schema for Stochastic Programs. *Annals of Operations Research* Under Review.

Fourer, R., Gay, D Kernighan, B. 1993. *AMPL: A Modeling Language for Mathematical Programming*. Scientific Press, San Francisco, CA.

Fourer, R. Goux, J.-P. 2001. Optimization as an Internet Resource. *Interfaces.* **31** 130-150.

Fourer, R. Ma, J. Martin, K. 2007. OSiL: An Instance Language for Optimization. Forthcoming in *COAP*. `http://www.springerlink.com/content/34jx62447n33w634/`.

GAMS. 2008. `http://www.gams.com/`.

Ma, J. 2005. Optimization services (OS), a general framework for optimization modeling systems. Ph.D. Dissertation, Department of Industrial Engineering & Management Sciences, Northwestern University, Evanston, IL.

Moré, J., T. Munson, J. Sarich. 2004. NEOS optimization server. `http://www-neos.mcs.anl.gov/neos/`.

OptiRisk Systems. 2004. Fortmp. `http://www.optirisk-systems.com/`.

Rosenbrock, H.H. 1960. An automatic method for finding the greatest or least value of a function. *Comp. J.*, **3** 175–184.

Skonnard, A., M. Gudgin, 2002. Essential XML Quick Reference. Pearson Education.

World Wide Web Consortium. 2007. XML Standard. Retrieved August 2, 2007, www.w3c.org/XML.

Examples Illustrating the OS Framework

EC.1. Calling an Optimization Solver

The goal of our research is to provide a framework for making optimization a service. The purpose of the COIN-OR OS project is to provide a reference implementation of the framework. Here we illustrate how the reference implementation of the framework is used to solve an optimization model in a distributed environment. In this illustration the client machine has a problem instance `rosenbrockmod.osil` (see Equations (1)-(4)) that is sent to the server for optimization. In each of the steps below the `OSSolverService` client reads a configure file at the command line and implements the options in the configure file. The reader may wish to refer back to Figure 15 while reading the material in this appendix.

Step 1: Use the `OSSolverService` method `getJobID()` method to get a job ID. In this step, and each of the steps that follow, a configuration file is given to the `OSSolverService`. The configuration file provides the necessary information for the `OSSolverService` to perform the required task. A call to the `OSSolverService` is illustrated below. In the call the configuration file is named `testRemotegetJobID.config`.

```
OSSolverService  -config ../data/configFiles/testRemotegetJobID.config
```

The options listed in the configuration file `testRemotegetJobID.config` are given below.

```
-serviceLocation http://gsbkip.chicagogsb.edu/os/OSSolverService.jws
-serviceMethod getJobID
```

This configuration file specifies two options. The first option, `serviceLocation`, specifies the location of the server performing the service. The second option, `serviceMethod`, is equal to `getJobID` and this tells the OSSolverService to request a job ID from the server. The server response to this request is a job ID that is a very long string involving IP numbers, time of day, etc. For illustration purposes, assume the job ID returned is `xyz123Dec3010AM`.

Step 2: Next, the model instance is submitted using the `send()` method. Again, a configuration file, `testRemoteSend.config` is passed to the `OSSolverService.`

```
OSSolverService  -config ../data/configFiles/testRemoteSend.config
```

The options in `testRemoteSend.config` are given below.

```
-serviceLocation http://gsbkip.chicagogsb.edu/os/OSSolverService.jws
-serviceMethod send
-osil ../data/osilFiles/rosenbrockmod.osil
-osol ../data/osolFiles/sendWithJobID.osol
```

The first line is again the service location of the server that will perform the optimization service. The second option is again the `serviceMethod` and tells the `OSSolverService` that the `send()` method should be used. The `send()` method requires an optimization instance and options for the solver. The third option in the configuration file is `osil` and this specifies the location of the optimization instance. Similarly the last option, `osol`, specifies the location of the file that contains the options provided to the solver. The `sendWithJobID.osol` file is

```
<osol>
  <general>
  <jobID>xyz123Dec3010AM</jobID>
  <contact transportType="smtp">
  kipp.martin@chicagogsb.edu
  </contact>
</general>
    <optimization>
     <other name="os_solver">ipopt</other>
    </optimization>
</osol>
```

The OSoL file provides the Job ID and an email address of the sender. The Web Service will send an email to this address notifying the sender when the job is complete. There is also an option telling which solver to use, in this case the COIN-OR solver `ipopt.` The result returned by server is `true` if the job is successfully submitted, `false` otherwise.

Step 3: The `knock()` service method is used to find the status of a job with a specific ID, or other jobs on the server and other summary statistics. Again, a configuration file is given to the `OSSolverService` on the client machine.

```
OSSolverService  -config ../data/configFiles/testRemoteKnock.config
```

The `testRemoteKnock.config` file is:

```
-serviceLocation http://gsbkip.chicagogsb.edu/os/OSSolverService.jws
-serviceMethod knock
-osplInput ../data/osplFiles/knock.ospl
-osol ../data/osolFiles/knock.osol
```

The configuration specifies the service location just as before and also specifies that the `serviceMethod` is `knock`. The `knock()` service method takes two arguments. The location of the file for the first argument is specified by the `osplInput` option. This argument provides information on the type of summary statistics that are desired. These options are specified using OSpL (Optimization Service Protocol Language). The OSpL file is

The `knock.ospl` file is:

```
<ospl>
     <processHeader>
          <request action="getAll"/>
     </processHeader>
     <processData/>
</ospl>
```

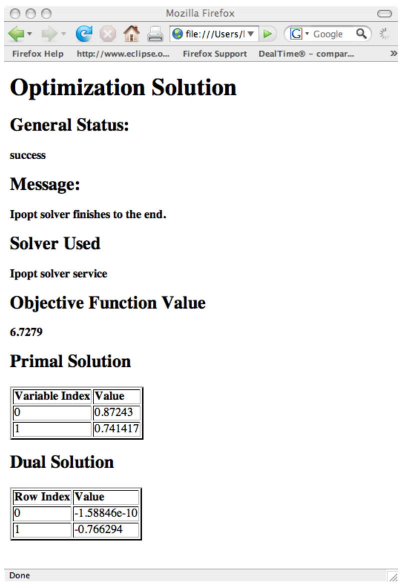
In this example, the key element in the OSpL file is the element `<request action="getAll"/>` that tells the server to return information on all of the jobs. The result of this request is very detailed information on all of the jobs solved by the server. This information is in `OSpL` format. The last option, `osol`, is the location of the file for the second argument for the `knock()` service method. For example, the user who wishes to override the `<request action="getAll"/>` request which can be quite voluminous, can specify a specific jobID in the `osol` option file and only information on that job is returned. This is illustrated below.

```
<osol>
     <general>
         <jobID>xyz123Dec3010AM</jobID>
    </general>
</osol>
```

A segment of the string that is returned in OSpL format is illustrated below. In this case we can see that the job with jobID `xyz123Dec3010AM` has finished.

```
<jobs>
     <job jobID="xyz123Dec3010AM">
          <state>finished</state>
          <serviceURI>http://192.168.0.219:8443/os/OSSolverService.jws
          </serviceURI>
          <submitTime>2008-01-30T13:18:58.769-06:00</submitTime>
          <startTime>2008-01-30T13:18:58.769-06:00</startTime>
          <endTime>2008-01-30T13:19:02.143-06:00</endTime>
          <duration>3.374</duration>
     </job>
</jobs>
```

Step 4: The last step is to retrieve the solution and display the results in a browser. The configuration file `testRemoteRetrieve.config` provides the necessary option values.



Optimization Solution

General Status:

success

Message:

Ipopt solver finishes to the end.

Solver Used

Ipopt solver service

Objective Function Value

6.7279

Primal Solution

Variable Index	Value
0	0.87243
1	0.741417

Dual Solution

Row Index	Value
0	-1.58846e-10
1	-0.766294

Figure EC.1 The optimization result displayed in a browser.

```
OSSolverService  -config ../data/configFiles/testRemoteRetrieve.config
```

The `testRemoteRetrieve.config` file is:

```
-serviceLocation http://gsbkip.chicagogsb.edu/os/OSSolverService.jws
-serviceMethod retrieve
-osol ../data/osolFiles/retrieve.osol
-osrl ./test.osrl
-browser /Applications/Firefox.app/Contents/MacOS/firefox
```

The first two options in the configure file are as in the previous steps. They specify the location of the Web service and the service method. In this case the `serviceMethod` is `retrieve` – i.e.

get the answer. The third line is the option that specifies the location of the options file. In this example, the options file contains the job ID that was assigned in Step 1, `xyz123Dec3010AM`, so that the server knows which job result to return. The file `retrieve.osol` is actually identical to the file `knock.osol` illustrated above. The `osrl` option on the fourth line tells the OSSolverService where to write the result of the optimization. In this case the solution result in the OSrL protocol is written into a file called `test.osrl` in the directory where the `OSSolverService` is executing. The last line is the location of the browser on the client machine. The optimal solution displayed in the browser is shown in Figure EC.1.

EC.2. Using the AMPL and GAMS with Optimization Services

Assume that the AMPL executable `ampl` (or `ampl.exe` on Windows) obtained from `www.ampl.com`, the OS `OSAmplClient`, and the test problem `hs71.mod` are all in the same directory. The AMPL model is `hs71.mod`. To solve this problem locally by calling the `OSAmplClient` from AMPL, first start AMPL and then execute the following commands. In this case the server invoked is the nonlinear COIN-OR sovler `Ipopt`.

```
# take in problem 71 in Hock and Schittkowski
# assume the problem is in the AMPL directory
model hs71.mod;
# tell AMPL that the solver is OSAmplClient
option solver OSAmplClient;
# now tell OSAmplClient to use Ipopt
option OSAmplClient_options "solver ipopt";
# now solve the problem
solve;
```

This will invoke `Ipopt` locally and the result in OSrL format will be displayed on the screen. In order to call a remote solver service, after the command

```
option OSAmplClient_options "solver ipopt";
```

set the solver `service` option to the address of the remote solver service.

```
option ipopt_options "service http://gsbkip.chicagogsb.edu/os/OSSolverService.jws";
```

In this case it is necessary that the `Ipopt` solver be part of the OS build on the server.

GAMS is used in a similar fashion. First, a problem is read from the GAMS library:

```
gamslib trnsport
```

Next, we tell GAMS to use the Optimization Services library

```
gams trnsport lp=os optfile=1
```

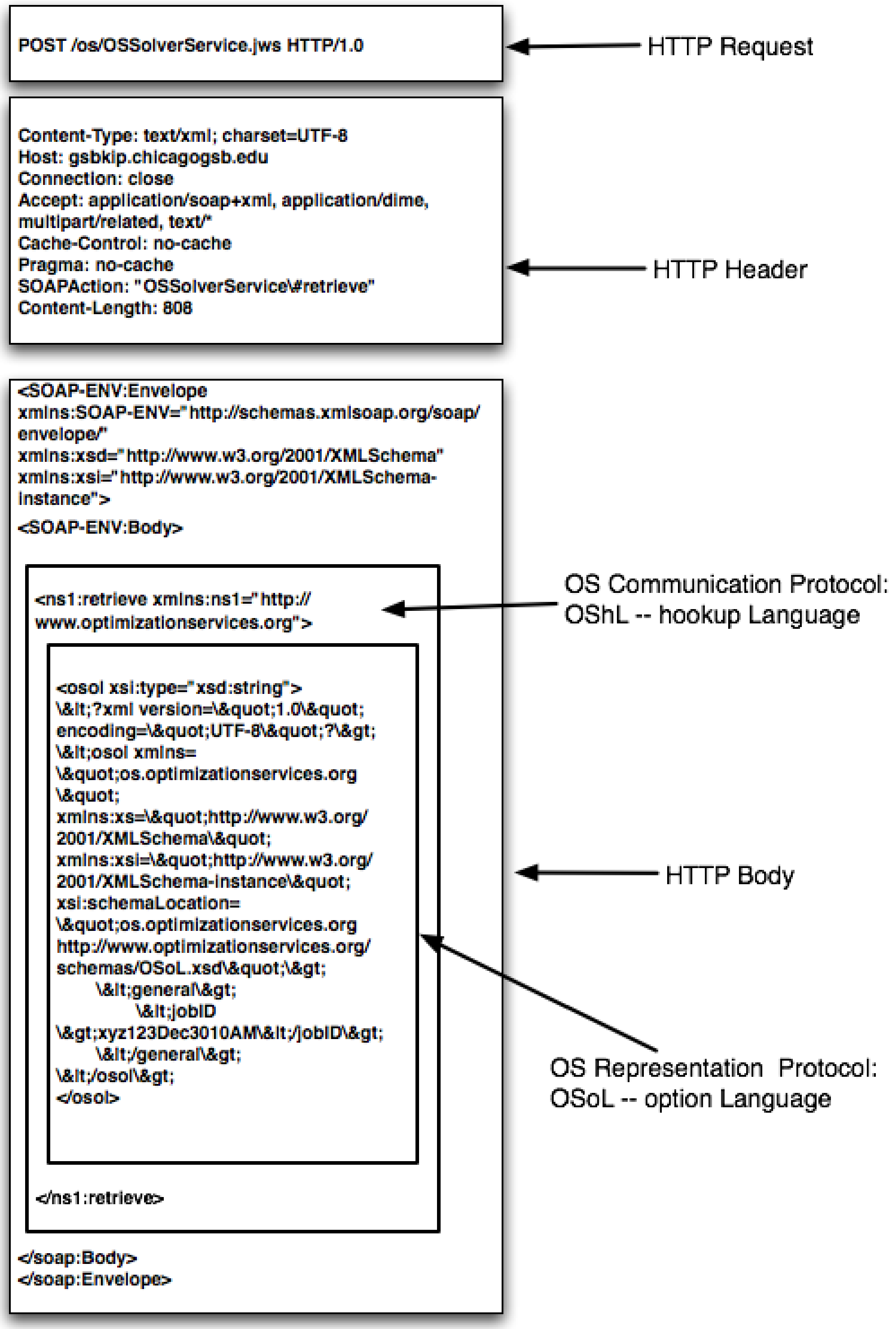
The option `optfile=1` tells GAMS to read the file `os.opt` for additional options. The `os.opt` file is

```
  writeosil osil.xml
  writeosrl osrl.xml
  service http://gsbkip.chicagogsb.edu/os/OSSolverService.jws
  solver clp
```

The first line instructs GAMS to output the model instanced to `osil.xml`. The second line instructs GAMS to write the solver output to `osrl.xml`. The `service` option specifies the location of the solver. Finally, the last option `solver` specifies that the COIN-OR solver `Clp` is to be used for model solution.

EC.3. Protocol Layers

In Figure EC.2 we illustrate in more detail what the user sees in Figure 6. In particular, note that there is an HTTP Request, an HTTP Header, and an HTTP Body. The body contains a SOAP envelop. Inside the SOAP envelop is the element `<retrieve>` which is a method specified in the the OShL communication protocol. This protocol contains another element `<osol>` which is a specific instance of the OSoL representation protocol.



```
POST /os/OSSolverService.jws HTTP/1.0
```

```
Content-Type: text/xml; charset=UTF-8
Host: gsbkip.chicagogsb.edu
Connection: close
Accept: application/soap+xml, application/dime,
multipart/related, text/*
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: "OSSolverService\#retrieve"
Content-Length: 808
```

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/
envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
<SOAP-ENV:Body>

  <ns1:retrieve xmlns:ns1="http://
  www.optimizationservices.org">

    <osol xsi:type="xsd:string">
    \&lt;?xml version=\&quot;1.0\&quot;
    encoding=\&quot;UTF-8\&quot;?\&gt;
    \&lt;osol xmlns=
    \&quot;os.optimizationservices.org
    \&quot;
    xmlns:xs=\&quot;http://www.w3.org/
    2001/XMLSchema\&quot;
    xmlns:xsi=\&quot;http://www.w3.org/
    2001/XMLSchema-instance\&quot;
    xsi:schemaLocation=
    \&quot;os.optimizationservices.org
    http://www.optimizationservices.org/
    schemas/OSoL.xsd\&quot;\&gt;
          \&lt;general\&gt;
                \&lt;jobID
    \&gt;xyz123Dec3010AM\&lt;/jobID\&gt;
          \&lt;/general\&gt;
    \&lt;/osol\&gt;
    </osol>

  </ns1:retrieve>

</soap:Body>
</soap:Envelope>
```

Figure EC.2 **OShL and OSoL Protocols Inside a SOAP Envelope Inside an HTTP Body.**