Web services for bioinformatics, Part 1

A framework for deploying bioinformatics applications as high-throughput web services on the NC BioGrid.

Mine Altunay
Daniel Colonnese
Chetna Warade

18 May 2004

This series describes the process of building, deploying, and using high-throughput web services for bioinformatics applications. This is meant to serve as a guide for development of software based on the Open-Bioinformatics Foundations software toolkits with packages such as BioPerl, BioJava, and BioPython. This article provides directions for how to deploy a service and present a new implementation of document-style web services extensions to the BioPerl module that will allow a wide range of existing applications to consume such services.

Web services for life sciences

The IBM alphaWorks article on web services for life sciences is an example set of web services that offers standard bioinformatics applications and demonstrates the technology (See Resources). The project is written in Java and mainly acts as a wrapper around existing bioinformatics applications. It allows researchers to search through the web service and obtain output through XML documents and to view intermediate steps throughout the request.

Bioinformatics research applications exist in various stages of completeness in many different languages. Furthermore, building a workflow from several different applications requires installing applications locally, copying input and output data by hand, and often modifying source code for adapting changes in output and input data format. However, when applications are exposed as web services, their executions can be coordinated fairly simply. When the services can communicate by producing and consuming compatible XML documents, then the process of orchestrating these services can be automated. Our proof-of-concept example Web Service for Basic Local Alignment Search Tool (BLAST) is hosted on the NC BioGrid (See Resources).

Efficiency, throughput, and workflow are major concerns when considering tools for comparative genomics. Web services hold the potential to address these concerns by allowing future applications to use on-demand processing power and best-of-breed components. Most of the
bioinformatics applications consist of very long workflows, which are mostly parallel. However for most workflows, only a small part of the workflow can be made parallel, whereas the rest of the workflow-chain just consists of stand-alone executables, each of which triggers the proceeding executables upon completion. Our work attempts to delegate the parts of the workflow that are able to be made parallel to the processing power of the NC BioGrid via web services and some extensions to the existing standard bioinformatics libraries. Therefore, any part of the workflow which may exploit the grid processing power will be submitted to the grid, and results will be returned in a form that can be integrated into the rest of the workflow.

This article requires an intermediate level of knowledge of the following technologies:

- Perl programming language
- BioPerl
- Open Grid Services Architecture
- Web services
- Web Services Description Language (WSDL) and bioinformatics applications
- Resources such as:
  - BLAST
  - ClustalW
  - Phylogenetic trees
  - PubMed
  - GenBank

**Document-style services**

There are several sets of overlapping Data Type Definitions (DTD) that are used to describe bioinformatics entities such as a Sequence, a Gene, or a BLAST input and output. The value of document style services is that, until a canonical form for data representation is established by the research community, web services using disparate DTDs can be taught to interoperate by transforming a document/literal message body with XSL stylesheets.

While many existing web services are Remote Procedure Call (RPC)-based, an increasing number of web services use document/literal encoding and communicate through exchanging XML documents. Document-style web services are useful in bioinformatics because many bioinformatics applications take so many parameters and produce such complicated structures that RPC becomes cumbersome to the programmer.

The Web Services Description Language (WSDL) file published by the web services server describes the data exchange format in XML Schema Definition (XSD) under the Types element, and the web service input and output are described as references to elements from the XSD schema.

Document-style web services let the programmers interpret in three ways:

1. Produce and send the literal XML
2. Produce Document Object Model (DOM) objects and send the XML produced by the DOM tree
3. Produce regular objects and serialize these objects into XML.

By using the DOM model for data exchange in document-style web services, a document can be produced simply by instantiating and combining a number of objects. However, for more complex documents, the code required to build a DOM tree can be unwieldy.

The Apache Axis tool can build a collection of Java objects from the XSD schema embedded in the WSDL. These objects contain get and set methods which allow the programmer to interact with them and have the ability to serialize themselves into XML. We have developed a similar tool, WSDL2Perl, for generating Perl objects from WSDL.

In order to use this tool to build objects, you must run the WSDL2Perl on the appropriate WSDL and then import and instantiate the generated objects. The web service can be invoked by instantiating the serializable objects. WSDL2Perl generates stubs that enable instantiation and invocation of web services through Perl objects. There are two possible modes: one where it consumes and produces literal XML, and the other where it consumes and produces objects generated by WSDL2Perl from the schema.

**WSDL2Perl: A developer tool for client-side bindings**

WSDL2Perl is a tool for facilitating document-style web services for Perl programming language. This tool requires SOAP::Lite v0.55 and is tested on the same. The WSDL2Perl tool supports only client side object and stub generation. WSDL2Perl allows invocation of web services developed and running on heterogeneous platforms (see section SOAP::Lite bugs).

The basic invocation of the WSDL2Perl tool looks like following,

**WSDL2Perl: Creating stubs and Perl data types from WSDL**

```
$ perl WSDL2Perl WSDL-file-URL
```

This command generates client bindings. The files created from this tool reside in the directory named as per the `targetNamespace` attribute. For each entry in the type section the WSDL2Perl tool generates a Perl object. The tool generates a stub class for each binding and generates a SOAP implementation (a locator class) for each service that supports SOAP transport protocol.

The Perl object generated from the WSDL type is named from the WSDL type. This object is the Perl module and hence has the extension .pm (for example, perlObjectName.pm.). For example, given the WSDL in Listing 2, WSDL2Perl generates the Perl module in Listing 3:

**Listing 2: Types**

```xml
<xsd:complexType name="phone">
  <xsd:all>
    <xsd:element name="areaCode" type="xsd:int"/>
    <xsd:element name="exchange" type="xsd:string"/>
    <xsd:element name="number" type="xsd:string"/>
  </xsd:all>
</xsd:complexType>
```
Listing 3: Generated Perl module

```perl
#!perl -w
use strict;
package Phone;
sub new{
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = {};   # this object
    my $self->{AREACODE} = undef;
    my $self->{EXCHANGE} = undef;
    my $self->{NUMBER} = undef;
    bless($self,$class);
    return $self;
}
sub areaCode{
    my $self = shift;
    if(@_) {
        $self->{AREACODE} = shift;
    }
    return $self->{AREACODE};
}
sub exchange{
    my $self = shift;
    if(@_) {
        $self->{EXCHANGE} = shift;
    }
    return $self->{EXCHANGE};
}
sub number{
    my $self = shift;
    if(@_) {
        $self->{NUMBER} = shift;
    }
    return $self->{NUMBER};
}
1;
```

PortTypes and bindings

In Java, the Service Definition Interface (SDI) is the interface that's derived from a WSDL's portType. This is the interface used to access the operations on the service. We could have one portType and two bindings, namely RPC and Document-style. Since document/literal changes what the interface looks like, we cannot use a single interface for both of these bindings, so we end up with two interfaces -- one named pt and another named bDoc -- and two stubs -- bRPCStub (which implements pt) and bDocStub (which implements bDoc). This convention is not applicable for WSDL2Perl as it is used for facilitating document-style web services only and the SOAP::Lite package supports easy use and access of RPC web services.

When a stub method implements the SDI, the method name is the same as the binding name. This stub translates its methods invocations into SOAP calls. This stub stands in as a proxy for the remote service; the stub behaves exactly as if the remote service were implemented in a local object. In other words, the user does not need to deal with the endpoint URL, namespace, or parameter arrays which are involved in dynamic invocation via the Service object and Call method.
The stub hides all that work from the user. All the stubs are contained in packages named after the ports.

**Listing 4: Example**

```perl
#!perl -w
package GridBlastSoap;
use strict;
use SOAP::Lite;
my $soap;
my $uri;
my $proxy;

sub new{
    my $proto = shift;
    my $class = ref($proto) || $proto;
    $uri = shift;
    $proxy = shift;
    my $self = {}
    bless($self,$class);
    return $self;
}

sub Call{
    my $result;
    my $onaction = $_[2];
    $soap = SOAP::Lite->new(uri="$uri",
        proxy="$proxy",
        on_action=>(sub{$onaction}),
        readable="1");

    if($_[1]){
        $result = $soap->call($_[0] => $_[1]);
    } else{
        $result = $soap->call($_[0]);
    }

    if($result->fault){
        print "GridBlastSoap: Fault ".$result->faultcode." has occurred ".
            $result->faultstring;
    }
    return $result;
}
```

A client program will not normally instantiate the stub directly. It would instead instantiate a service and call a get method which would return a stub. The service is derived from the service clause in the WSDL file. WSDL2Perl generates one object from a service clause. For example, given the WSDL in **Listing 5**, WSDL2Perl generates the object named from a service, as shown in **Listing 6**:

**Listing 5: Services**

```xml
<service name="GridBlast">
    <documentation>The first Web Service for a grid enabled BLAST brought to you by IBM and NC State Univ.</documentation>
    <port name="GridBlastSoap" binding="t:GridBlastSoap">
        <soap:address location="http://bluejay001.ncbiogrid.org/cgi-bin/GridBlast.cgi"/>
    </port>
</service>
```

Web services for bioinformatics, Part 1
Listing 6: Object named from a service

```perl
#!/perl -w
use strict;
package GridBlast;
use GridBlastSoap;

my $uri = "http://bluejay001.ncbiogrid.org/";
my $proxy = "http://bluejay001.ncbiogrid.org/cgi-bin/GridBlast.cgi";

sub getGridBlastSoap{
  my $soapservice;
  if($_[1]){
    $soapservice = GridBlastSoap->new($uri, $_[1]);
  }else{
    $soapservice = GridBlastSoap->new($uri, $proxy);
  }
  return $soapservice;
}
```

The Perl object generated defines a get method for each port listed in the Service element of the WSDL. The Service object will by default make a stub that points to the endpoint URL described in the WSDL file, but the user may specify a different URL by passing the desired URL as an argument to the get method.

**Extensions to standard bioinformatics libraries (BioPerl)**

The most popular open source platform for bioinformatics applications are the libraries released by the Open Bioinformatics Foundation. The largest of these libraries is BioPerl. BioPerl has several modules, each of which can be used to create an application workflow in Perl. BioPerl library enables use of a collection of pre-built software objects for the applications. Most of the standard analysis tools such as BLAST and ClustalW are already included in the BioPerl libraries, so that all the end user needs to do is simply include those modules in the workflow and make the necessary function calls. For example, in order to do a BLAST, a well-known standard homology search application, there are two different modules in the BioPerl library: StandAloneBlast and RemoteBlast. StandAloneBlast assumes that users have the executable on their host machine and passes the arguments to the executable, which returns the result in a human-readable format. RemoteBlast makes function calls to a BLAST executable at the National Center for Biotechnology Information (NCBI) website, via http, so that users do not have to install the actual binaries on their host machine.

Our extension to the BioPerl libraries is the WebServiceBlast module, which allows users to exploit remote processing power of the grid via web services. This module has the same signature as the RemoteBlast module and returns the result of the homology search tool, (known as BLAST), in a similar format. Therefore, this extension does not assume or bring any modifications to the existing BioPerl modules. BioPerl libraries' SearchIO modules may easily parse resulting BLAST output files. Therefore, a bioinformaticist who wants to exploit remote grid processing power simply needs to invoke WebServiceBlast instead of the RemoteBlast module.

WebServiceBlast module objects are inherited from Bio::Root::Root and Bio::Root::IO. The included modules from the BioPerl library are:
• Bio::Root::Root
• Bio::Root::IO
• Bio::SeqIO
• Bio::Tools::BPlite
• Bio::SearchIO.

IO::String and SOAP::Lite are the other necessary modules from the CPAN libraries.

The main functionality of the WebServiceBlast module is to create an XML document from the user arguments. The WebServiceBlast allows invocation of the web services through the Perl objects that the WSDL2Perl tool generates. The resulting BLAST output report is returned to the workflow in XML format. This module also lets users pass the name of their input files containing genomic sequences, so that input sequences will be automatically saved into XML documents. In cases where users need to submit more than one sequence to fully exploit the processing power of the grid, WebServiceBlast creates only one XML document. This document contains all the information necessary for each submitted sequence, such as the sequence identification tag, sequence description, length, and nucleotide chain.

On the NC BioGrid, upon completion of the BLAST execution, the WebServiceBlast module automatically saves the results into the desired directory on the host machine and provides XML BLAST with output reports. BioPerl's SearchIO module can easily parse these reports. The following is an example of the client code that is making calls to the WebServiceBlast module.

Listing 7: Running WebServiceBlast

```perl
use Bio::Perl;
use Bio::Tools::Run::WebServiceBlast;
use Bio::SearchIO;

my $prog='blastn';
my $db='swissprot';
my $e_val='1e-10';
my $MATRIX='BLOSUM62';

#Object construction from WebServiceBlast module
my @params=('-prog'=>$prog, '-data'=>$db,
'-expect'=>$e_val, '-readmethod'=>'Blast', '-matrix'=>'BLOSUM62');
my $factory=Bio::Tools::Run::WebServiceBlast->new(@params);

#Function call to NC BioGrid via Web Services
my $res=$factory->submit_blast("sample.fasta");

#Preparation of output files
push @outfile, "/home/maltuna/ncgridtools/Bioperl/bioperl-1.2.1/out1";
push @outfile, "/home/maltuna/ncgridtools/Bioperl/bioperl-1.2.1/out2";

$factory->save_output($res, @outfile);

#An example of how to parse blast output reports in XML format in
#SearchIO module of BioPerl library
my $searchio = new Bio::SearchIO (-format =>'blastxml','
```

Web services for bioinformatics, Part 1
SOAP::Lite bugs

A SOAPAction header entry is a hint to the web service indicating the name of the method to be called. The SOAP specification does not specify the format of the SOAPAction HTTP header beyond indicating that it is a URI, so SOAP::Lite and .NET have different default formats. By default, SOAP::Lite creates a SOAPAction header that looks like [URI]#[method]. A CGI-based SOAP server expects a SOAPAction header that looks like [URI]#[method]; however, .NET requires that the SOAPAction header to look like [URI]/[method]. We override the default SOAP::Lite implementation of on_action() with the attribute soapAction for the particular SOAP operation.

Summary

Document-style web services allow for greater application interoperability and performance, although there are some trade-offs made for complexity and storage requirements when using such services. Apache AXIS has added support for document-style services in Java applications. Microsoft uses document style for .NET services, and the specification for Grid Services in OGSA mandates the use of document-style services. Document-style web services are presented here as extensions for the Perl language, and by using these extensions and expanding BioPerl to consume document-style services, this project is paving the way for these technologies to interoperate.

Acknowledgements

This paper describes the joint work of the Extreme Blue team Summer 2003, Fungal Genomics Lab at NC State University and the North Carolina Biogrid. Our team has set up a framework for deploying bioinformatics applications as high-throughput web services on the North Carolina BioGrid. The intern team consists of: Mine Altunay (maltuna@unity.ncsu.edu), Daniel Colonnese (dcolonn@ncsu.edu), Chetna Warade (warade@us.ibm.com), and Lindsay Wilber (WilberL04@darden.virginia.edu). The team was advised by members of the IBM Life Sciences Group, including Virinder Batra (batra@us.ibm.com), Madhu Gombar (mgombar@us.ibm.com), Rick Runyan (runyan@us.ibm.com), Prasad Vadlamudi (prasadv@us.ibm.com) and Doug Brown (debrown@unity.ncsu.edu).

© Copyright IBM Corporation 2004

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)