The ERATO Systems Biology Workbench: Architectural Evolution

Michael Hucka, Andrew Finney, Herbert Sauro,
Hamid Bolouri, John Doyle, Hiroaki Kitano

ERATO Kitano Systems Biology Project
Control and Dynamical Systems, MC 107-81
California Institute of Technology, Pasadena, CA

sysbio-team@caltech.edu

ABSTRACT

Systems biology researchers make use of a large number of different software packages for computational modeling and analysis as well as data manipulation and visualization. To help developers easily provide the ability for their applications to communicate with other tools, we have developed a simple, open-source, application integration framework, the ERATO *Systems Biology Workbench* (SBW). In this paper, we discuss the architecture of SBW, focusing on our motivations for various design decisions including the choice of the message-oriented communications infrastructure.

1. INTRODUCTION

The explosion of interest in systems-level modeling and analysis of biochemical networks has been accompanied by the development of a diversity of software tools. While this diversity of resources is welcome, it also has had unwelcome side-effects. One is a duplication of efforts by different research groups attempting to incorporate similar capabilities into their simulation/analysis tools. In an effort to make it more attractive for package developers to share rather than reimplement resources, we have developed the ERATO *Systems Biology Workbench* (SBW), an open-source, application integration environment. Our aim has been to create a framework so simple that software developers find it easier to build in an SBW interface than to recreate functionality available in other tools. By doing so, we hope developers can concentrate on developing best-of-breed solutions in the areas where they have special expertise.

SBW uses a portable broker-based architecture that enables applications (potentially running on separate machines) to learn about and communicate with each other. The communications facilities allow heterogeneous packages to be connected together using a remote procedure call mechanism; this mechanism uses a simple message-passing network protocol and allows either synchronous or asynchronous invocations. The interfaces to the system are encapsulated in client libraries for different programming languages (currently C, C++, Delphi, Java, and Python, with more anticipated), but the protocol is open and small, and developers may implement their own interfaces to the system if they choose.

Frameworks for integrating disparate software packages are certainly not new. Some of the more well-known application frameworks are Cactus Code [1], ISYS [33], Net-Solve [11], and Ninf [29]; some of the more well-known middleware integration frameworks are CCAT [8], CORBA [38], DCOM [13], ILU [24], Jini [5], Nexus [14], and PVM [15]. When we began work on SBW, we intended to begin with an existing framework such as CORBA and simply augment it with additional facilities. But after examining a number of alternatives, we were forced to conclude that existing systems did not provide an adequate combination of the features we believed were needed. These features include: simplicity in both the API and the data exchange protocol, support for major programming and scripting languages and the seamless interaction between modules written in different languages, support for dynamically querying modules for services they offer, portability to both Windows and Linux (with a clear ability to be ported to other platforms), and free, unrestricted availability of implementations (with source code) for all platforms.

In the following sections, we summarize some of the key architectural design decisions and how we arrived at them over the course of SBW's evolution. We also explain why we developed a custom message-passing protocol over a number of alternatives such as SOAP [7], XML-RPC [41] and MPI [20].

2. ARCHITECTURAL EVOLUTION
2.1 Goals for User Interaction

We began the project with the general goal of enabling interaction between a number of existing simulation and analysis tools. These tools were *BioSpice* [4], *DBsolve* [19], *E-Cell* [37], *Gepasi* [25], *Jarnac* [30], *StochSim* [9], and *Virtual Cell* [32]; they are packages in common use by researchers studying a variety of topics, including the modeling of metabolic pathways, cell signaling pathways, gene regulation, and many others. We also wanted to provide the ability for new tools (such as *ProMoT/DIVA* [17]) to be easily introduced.

Software tools may come in the form of libraries implementing certain specialized algorithms, or full-fledged applications with complete user interfaces. Our initial vision for SBW centered around a stand-alone, GUI-based, central management tool that would display to the user a list of all the packages available for interaction within the framework. In this system, a user would begin a session by starting the management tool and then selecting a package from the list.

The tool would start the requested package and bring forth its user interface, if it had one. If a package did not have its own interface, we envisioned the management tool would provide a forms-based interface that could be constructed based on the package's description. When the user elected to send the results of one package to another, we envisioned the management tool as an intermediary, receiving and holding the data and letting the user pick the next package from the list of known packages.

After some initial efforts in this direction, we abandoned the idea of a central manager. While it had a certain organizational appeal, we discovered that the management tool created an awkward bottleneck in the user's interactions between packages, interrupting the user's flow between tools. A better scheme would be to have the system act as an invisible service to the collection of SBW-enabled packages installed on a user's computer, *giving center stage to the tools themselves rather than the framework*. We redesigned SBW accordingly. In the new scheme, users typically start the first application as they would any other program; they do not need to do anything special to start SBW itself. Moreover, SBW is not a controller in the system—the flow of control is entirely determined by what the individual modules and the user do.

Figure 1 shows an example of using a collection of SBW-enabled software modules. The upper left-hand area in the figure (partly covered by other windows) shows an SBW-enabled version of JDesigner [31], a visual biochemical network layout tool. This module's appearance is nearly identical to that of its original non-SBW-enabled counterpart, except for the presence of a new item in the menu bar called **SBW**. This is typical of SBW-enabled programs: the SBW approach strives to be minimally intrusive.

SBW software components (called SBW *modules*) can come in different forms. A module may be primarily computational and lack a GUI, or it may be a computational module having its own GUI, or it may consist solely of a GUI designed to control other tools. In the example shown in the figure, the user has created a network model in JDesigner, then has decided to run a time-series simulation of the model. To do this, the user pulled down the **SBW** menu (not shown in the figure) and selected one of the options, *Jarnac Analysis*, to invoke the SBW-enabled simulation program *Jarnac* [30]. This brought forth a control GUI, shown underneath the plot window in the lower right-hand area of Figure 1; the user then input the necessary parameters into the control GUI to set up the time-series simulation, and finally clicked the **Run** button in the GUI to start the simulation. The control GUI used SBW calls to instruct the simulation module (Jarnac) to run with the given parameters and send the results back to the controlling GUI module, which then sent the results to a plotting module.

This example scenario illustrates the interactions involved in using SBW and four sample modules: the visual JDesigner, a control GUI for time-series simulations, the computational module Jarnac, and a graphical plotting module. The basic process described above can be extended to any number of modules in a system; for example, the user could have

chosen to send the results of the time-series simulation to another module for further analysis, instead of plotting it directly. The style of interaction described here, with different modules taking center stage in turn, is meant to provide context for what the user is doing at any given moment. The underlying assumption is that each module/application is itself best suited to providing that context, lending a sense of familiarity and *place* to the task it supports.

2.2 Additional Technical Requirements

In addition to the goal of supporting the style of user interaction described above, we also had the following requirements for the framework:

- *Simplicity*: The framework must be simple enough that interested developers can use it in their projects with a minimum amount of learning and coding effort.

- *Component modularity*: As new tools and methods are developed, it must be possible to implement them as modules that can be hooked into the existing framework without having to modify the framework itself.

- *Language interoperability*: The framework must support the interaction of modules written in different programming languages.

- *Free distribution*. All interested users must be able to obtain both SBW and its source code for free. Any software that is incorporated into SBW and distributed with it, such as GUI widgets or object libraries, must itself be free of licensing fees or restrictions on redistribution. (This is only a requirement on SBW itself, and not on modules built for SBW or other software developed using SBW.)

- *Portability*. The framework must be portable to Microsoft Windows (NT, 2000, XP) and Linux initially, and clearly be portable to other platforms in the future.

- *On-demand plug-in loading*: Modules that implement particular capabilities should not have to be pre-loaded into SBW every time it is started; instead, the system should be data- and task-driven and dynamically load modules on an as-needed basis. This helps keep the size of the running system to a minimum.

- *Support for distributed computing*: The user should have control over where processes are executed and the ability to interact with remote services.

Some of these requirements immediately eliminated from consideration certain existing frameworks. For example, the portability and free availability requirements eliminated DCOM [13], which for all practical purposes is limited to Microsoft Windows platforms, and the language interoperability requirement eliminated Jini [5] and Java RMI [23], which are only practical if all applications use Java.

CORBA [38] and similar frameworks such as ILU [24] could meet the goals above and serve to implement SBW. We have been reluctant to use CORBA as the basis of SBW for two main reasons. First, to interface software packages written in different languages nearly always requires having to
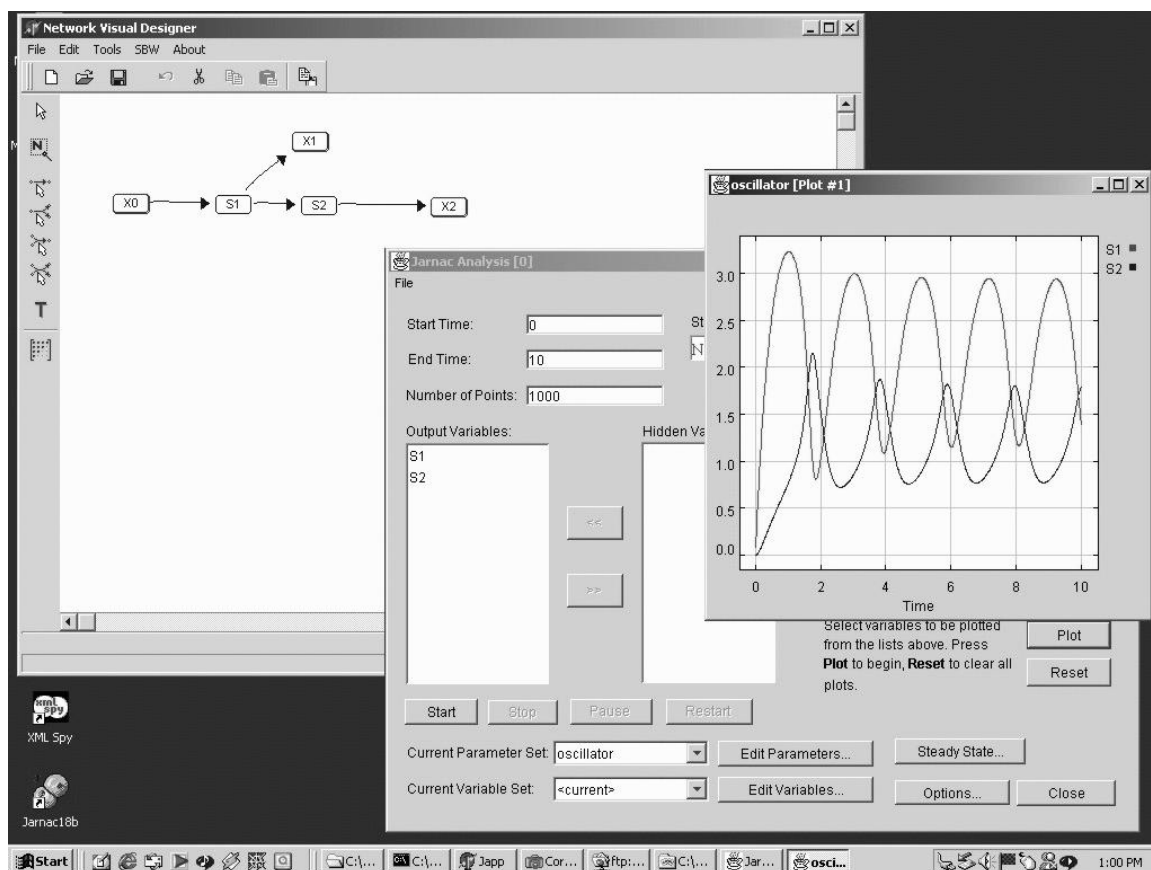
Figure 1: Example of multiple applications interacting through SBW.

use different CORBA implementations, because there are no open-source implementations that provide bindings to a sufficiently wide variety of programming languages. This has a number of implications. For example, we would have to test the compatibility of (and if necessary, provide adapters for) all the CORBA distributions we support for SBW, for all language combinations that SBW supports. The difficulty in doing this lies in the sheer size and complexity of CORBA implementations. Second, the complexity of CORBA is simply unsuitable for a software system that will, at least initially, be largely used by academic researchers and university-based programmers. Many of the innovations in biological modeling come from these environments and take the form of single modules implemented by these researchers and programmers. Their time and resources are too limited to require them to read and understand a thick manual on CORBA before they can interface their module into a framework. Further, many programmers are predisposed to discount CORBA as slow, big and inefficient; regardless of the validity of these impressions, it would be too difficult for us to attempt to overcome them.

Notwithstanding these issues, we are not in principle opposed to providing a way for CORBA users to interact with SBW. We are currently designing an interface that will provide a CORBA bridge to SBW for those developers who prefer to use this technology.

2.3 The SBW Architecture

A number of different architectures can support the style of user interaction described in Section 2.1 and the technical requirements listed in Section 2.2. True to the old programming adage, "plan to throw the first version away," our first implementation taught us lessons that forced a redesign and the development of a new version.

2.3.1 The Initial Version

Our first iteration on the SBW architecture was oriented around a two-layer architecture. The bottom layer (which we called the *Biological Modeling Framework* [18]) was a general software framework that provided basic scaffolding supporting a modular, extensible application architecture, as well as a set of useful software components (such as GUI tools) that could be used as black boxes in constructing a system. SBW comprised the top layer; it consisted of pluggable modules ("plug-ins") that collectively implemented what users experienced as the "Systems Biology Workbench". Figure 2 illustrates the general organization of the architecture.

We demonstrated a working implementation of SBW using this architecture in the winter of 2001. This prototype consisted of an SBW core plug-in implemented in Java, a plug-in plotting module, a plug-in that provided as an interface to a visual pathway layout tool, a plug-in that provided an interface to an ODE-based simulator, and a plug-in GUI
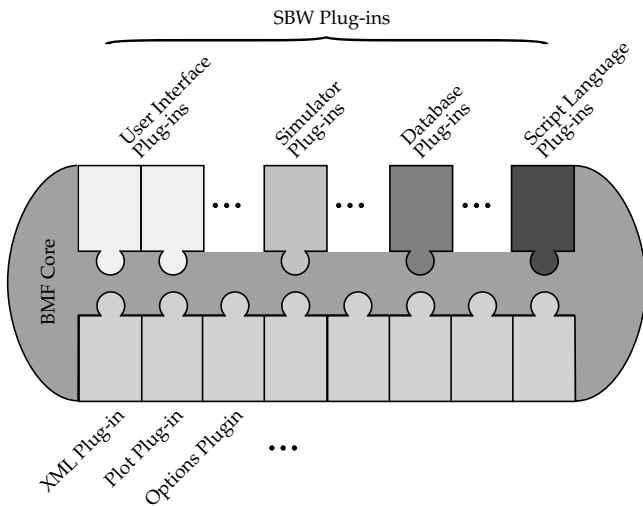
3

Figure 2: The organization of the initial SBW architecture, in which the system consisted of plug-ins that provided interfaces to different facilities or external tools.
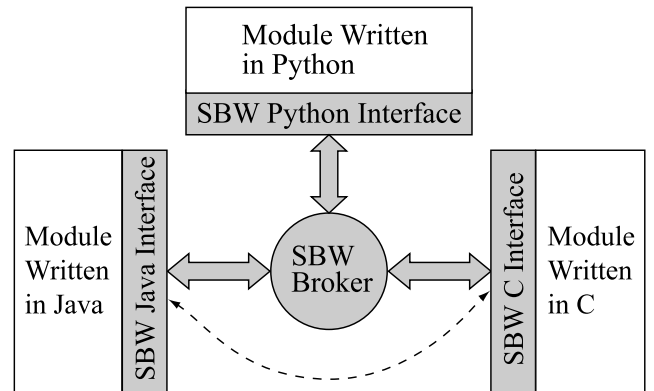

Figure 3: The overall organization of the current, second-generation SBW system. Gray areas indicate SBW components (libraries and the broker). To individual modules, communications appear to be direct, although they actually pass through the broker. A planned optimization is to use shared memory calls to implement direct, module-to-module communications scheme when the modules are collocated on the same machines. Such details are hidden behind the SBW interfaces and will not require any code changes to the modules themselves.

control tool that controlled the simulator. Although the architecture supported much of the functionality we sought, we also learned that it was not ideally suited to our goals. There were two main problems. First, most of the modules in the system turned out to be stand-alone applications, not libraries. Although the plug-in organization worked, the requirement that each stand-alone application have a matching plug-in interface introduced an extra layer that had questionable value and high maintenance cost. Second, the plug-in organization meant that a library written in a language other than Java had to be interfaced to the core using Java's native language interface. This introduced a performance penalty as well as considerable complexity.

2.3.2 The Second Version

Based on our experiences with the initial architecture described above, we have developed a new system that avoids the centralized plug-in organization. This new architecture (used in the the released version of SBW) replaces the plug-in core of Figure 2 with a broker whose job is to act as an intermediary that enables communications between separate software components ("modules"). A software module may be a client, or a provider of services, or both, and it may be a stand-alone application or a library. In all cases, the module connects to the rest of the system through the SBW client library. This library provides an API that a programmer can use to interact with SBW. Figure 3 illustrates the overall system organization.

Broker architectures are relatively common and are considered to be a well-documented software pattern [10]; they are a means of structuring a distributed software system with decoupled components that interact by remote service invocations. In SBW, the remote service invocations are implemented using *message passing*, another tried and proven software technology. Message-passing systems implement inter-component communications as exchanges of structured data bundles—messages—sent from one software entity to

another over a channel. Some messages may be requests to perform an action, other messages may be notifications or status reports. Because interactions in a message-passing framework are defined at the level of messages and protocols for their exchange, it is easier to make the framework neutral with respect to implementation languages: modules can be written in any language, as long as they can send, receive and process appropriately-structured messages using agreed-upon conventions.

Message-passing schemes traditionally suffer from one drawback: as the messaging scheme becomes more elaborate, it becomes more complicated and error-prone to program at the application level. From the application programmer's point of view, it is preferable to isolate communications details from application details. For this reason, we provide two levels of programming interfaces in SBW: a low-level API that consists of the basic operations for constructing and sending messages, and a high-level API that hides some of the details of constructing and sending messages and provides ways for methods in an application to be "hooked into" the messaging framework at a higher level. The APIs are discussed further in Section 3.

2.3.3 Comparisons to Other Architectures

The architecture of SBW has a number of similarities to that of DCOP, the Desktop Communication Protocol used in the KDE desktop environment for Linux [36]. DCOP allows applications to pass data between them by serializing the data into messages, and applications manipulate the messages using a data stream object. A broker program dispatches messages between applications. Libraries implement the DCOP APIs for different supported languages, and an optional interface definition language (IDL) mechanism is

4

available for generating interface code for client applications. Applications may choose to avoid using the IDL and instead may specify the signatures of remote procedures using simple strings such as ``int mymethod(string arg1, double arg2)''. SBW likewise uses strings to define the signature of each method made available for a service in an application; in addition, in the case of object-oriented languages such as Java and Python, the SBW client libraries use the signature strings to construct proxy objects corresponding to each service. This provides a natural, object-oriented interface to the services provided by a module, rather than the basic, procedural function-call interface provided by DCOP.

SBW also has architectural similarities to ISYS [33]. This system provides a generalized platform into which components may be added in whatever combination the user desires. It uses a bus-based communications framework that allows components to interoperate without direct knowledge of each other, by using a publish-and-subscribe approach in which components place data on the bus and other components can listen for and extract the data when it appears. ISYS components include graphical visualization tools and database access interfaces. The main differences between the SBW and ISYS architectures are the following. First, SBW uses point-to-point communications instead of the bus architecture of ISYS. The bus likely provides a more flexible interface than a direct, point-to-point organization, at the cost of somewhat higher implementation complexity. Second, ISYS interfaces are defined entirely in Java, whereas in SBW, they can be written in any language. ISYS modules written in other languages must be wrapped with a Java interface layer to let them interact with the rests of the system. By contrast, in SBW, interfaces are expressed in terms of messages, and the message-handling code can be implemented in native-language API libraries, avoiding the need for Java wrappers.

2.3.4 Comparison to Other Message-Oriented Frameworks

The communications infrastructure of SBW could have been implemented in a number of ways. Our main criteria for choosing a suitable message-passing scheme were: performance, support for data types needed for SBW, simplicity, and portability. We examined a number of existing frameworks including XML-RPC [41], SOAP [7], MPI [20], and Java RMI [23], but in the end decided to implement our own simple communications framework. In Appendix A, we present a comparison of SBW's communications implementation and a specific viable alternative, XML-RPC, as an example of the decision processes that drove the design choices in SBW.

3. PROGRAMMING WITH SBW

A module may make a set of operations available programmatically to other software modules. In SBW, sets of methods or functions are grouped into one or more *services*. A service is an interface to a resource inside a module. This interface consists of a collection of methods that encapsulate access to some set of functionality.

A given module may implement zero or more services. More than one module may provide similar services, so it is useful to be able to group services into categories. SBW supports optional *service categories* of this kind explicitly in the APIs, but it is entirely up to applications to decide how to structure and manage the categories.

Fig 4 illustrates schematically the SBW client library interface. We strove to develop a high-level API for SBW that provides a natural interface in each of the different languages for which we have implemented libraries so far. By "natural", we mean that it uses a style and features that programmers accustomed to that language would find familiar. For example, in Java, the high-level API is oriented around providing SBW clients with proxy objects whose methods implement the operations that another application exposes through SBW.

We believe that by using the libraries and high-level API, application developers will find it relatively easy to introduce SBW interoperability into their software. The facilities provided by the high-level SBW API can be summarized as follows:

1. *Dynamic service and module discovery*: These are facilities for querying SBW about the services, service categories, and modules running at any given moment.

2. *Module, service and method registration*: The broker keeps a registry that other modules can use to search for particular modules and services. The registration facilities allow a module to record with the Broker the services and methods that the module provides, as well as to provide help text describing the service methods and the command that should be used to start up the module on the fly.

3. *Remote method invocation*: This enables one module to invoke a service method in another module. As mentioned above, SBW provides both a basic low-level API, and a higher-level API that allows invoking methods in a way that is "natural" for the particular programming language being used.

4. *Data serialization*: All method invocations involve passing messages between modules, which requires packing data into message streams. (In Java, it is possible to abstract these details completely and introduce proxy objects that completely encapsulate operations on remote modules. Unfortunately, this is not possible in all languages; in those cases, some aspects of data serialization must be exposed to the application.)

5. *Exception handling*: These are facilities for dealing with exceptional conditions signaled by modules.

6. *Event notification*: Certain activities in SBW, such as the addition or shutdown of an instance of a module, generate events that are used to notify all other modules of the changes.

As an example of how simple the API is to use in practice, the following Java code demonstrates how one might invoke a simulation module that implements a handful of methods. The methods are part of a service named "simulation" provided by the module, and the module is named "edu.caltech.simulator" in this example.
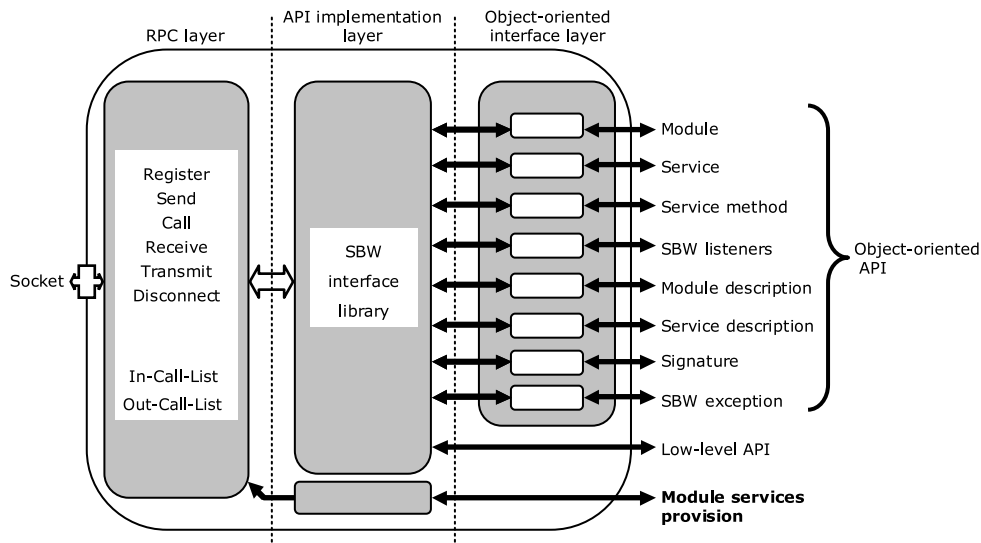
Figure 4: Illustration of the components of the SBW client library.

```
// Define the service interface for the Java compiler.
interface Simulator
{
  void loadSBML(string);
  void setTimeStart(double);
  void setTimeEnd(double);
  void setNumPoints(integer);
  double[] simulate();
}

public double[] runSimulation(String modelDefinition,
                              double startTime,
                              double endTime,
                              integer numPoints)
{
  try
  {
    // Start a new instance of the simulator module.
    Module module
      = SBW.getModuleInstance("edu.caltech.simulator");

    // Locate the service we want to call in the module.
    Service srv = module.findServiceByName("simulation");
    Simulator simulator
      = (Simulator) srv.getServiceObject(Simulator.class);

    // Send the model to the simulator and set parameters.
    simulator.loadSBML(modelDefinition);
    simulator.setTimeStart(startTime);
    simulator.setTimeEnd(endTime);
    simulator.setNumPoints(numPoints);

    // Run the simulation and return the result.
    return simulator.simulate()

  } catch (SBWException e) {
    // Handle problems here.
  }
}
```

4. SUMMARY

Our goal in developing the ERATO Systems Biology Workbench has been to create a simple, open-source integration environment that will allow developers to easily provide the ability for their applications to communicate with other tools. SBW uses a straightforward broker-based architecture that allows applications (potentially running on separate, distributed computers) to communicate via a simple network protocol. The interfaces to the system are encapsulated in client-side libraries that we provide for different programming languages (currently C, C++, Java, Delphi and Python, with more anticipated in the future). By providing a common, free framework for linking different packages together, we hope to enable a new synergy of functionality that will allow greater exchange of models and results in all areas of computational biology.

A number of SBW-enabled modules are already available. These include: (1) a visual biochemical network designer [31]; (2) an ODE-based network simulator [30]; (3) a simple SBW status monitor that sits in the Microsoft Windows *tray* and can be used to check on running modules; (4) a plotting module; (5) an SBML [22] reader, validator and model query module; (6) a stochastic simulation module based on the work of Gibson [16]; and (7) an optimizer based on the core Gepasi [25] algorithms. More modules are under development by our group and others.

Acknowledgments

APPENDIX
A. MESSAGE-PASSING IN SBW: EXPERIMENTS WITH ALTERNATIVES

In this section, we support our decision to develop a novel messaging scheme in SBW by presenting a detailed comparison between SBW's messaging framework and XML-RPC [41] (a simpler variant of SOAP [7], a popular Internet protocol). We chose XML-RPC as a test case for several reasons: it is a simple and easily-understood protocol, im-

plementations are freely available for a large number of programming languages, and it is similar to SOAP, yet simpler and smaller, which means that the results of performance tests using XML-RPC reflect the best results that one could expect from using SOAP.

A.1 Summaries of the Two Systems

A.1.1 SBW-RPC

The message-passing system implemented in SBW (which we call *SBW-RPC*) is a relatively simple scheme that encodes data values as bytes and sends them in a stream across a socket connection. Each data item is preceded with a tag that identifies its type. The messaging system supports the following common and useful data types:

Byte	An 8-bit quantity, equivalent to a Java `byte`
Boolean	0 (false) or 1 (true)
Integer	32-bit signed number, equivalent to Java's `int`
Double	64-bit floating-point number (IEEE 754 format)
String	Sequence of characters equivalent to `char *` in C
Array	Multidimensional homogeneous block of data
List	Heterogeneous sequence of data elements

The operations in the basic API center around blocking and non-blocking remote procedure calls. The blocking version (`call`) invokes a specific method in a specific module, handing it arguments serialized into a message data stream. The call waits until the method on the remote module returns a value. The non-blocking version (`send`) is similar, except that it does not wait for a return value. SBW-RPC also provides a means for returning exception codes to a caller.

We were reluctant to invent a custom message encoding format, given that a number of alternatives already exist. We examined numerous well-known candidates, including Sun RPC XDR [34], DCE NDR [27], and the Java Object Serialization Specification [35]. However, from the standpoint of our requirements, each of these alternatives suffered from deficiencies. For example, both XDR and NDR are untagged formats, which means that two parties communicating data must know ahead of time the exact structure being sent, and Java serialization has substantial (and for our purposes, unnecessary) baggage for dealing with object classes.

A.1.2 XML-RPC

XML-RPC [41] is a remote procedure calling protocol that uses HTTP as the transport and XML as the message encoding. The messages are in plain text, and consist of tagged fields for the name of the remote procedure being called and the arguments supplied. The textual nature of the messages makes debugging simpler. XML-RPC supports a number of data types:

Boolean	0 (false) or 1 (true)
Integer	32-bit signed number, equivalent to Java's `int`
Double	64-bit floating-point number
String	Sequence of ASCII characters
DateTime	Date/time in ISO 8601 format
Base64	Base64-encoded [6] binary data
Array	A heterogeneous sequence of elements
Struct	A heterogeneous sequence of elements, each of which contains named item-value pairs

Arrays in XML-RPC differ from those in SBW-RPC in that they can be heterogeneous. Each element in an array is prefixed with a type specifier. It is also worth noting that in XML-RPC, the format of floating point numbers is not specified as being, e.g., IEEE 754 format, and the numbers themselves are communicated as strings. (The size specification (64-bit) refers to the space that should be allocated to the data type in a program, not to the size of the data object as encoded in XML.) Finally, there is no representation for infinity, negative infinity, or "not a number". The range of allowable values is implementation-dependent and not specified.

An XML-RPC transaction is synchronous and uses HTTP as the protocol. The call returns a response message to the caller; this message may be a fault/exception.

A.2 Performance Comparison of SBW-RPC versus XML-RPC

Data exchange rates can be an important issue for biological simulation and analysis. If the time required to transmit data between software tools is too high, a framework such as SBW will be rejected by software tool developers as being too inefficient. We therefore placed high value on providing a fast communications mechanism in SBW.

A.2.1 Experimental Methods

We tested Java implementations of both the XML-RPC and SBW-RPC systems. The performance tests consisted of timing how long it took each messaging implementation to exchange a certain number of messages. The tests of the XML-RPC and SBW-RPC versions both used the same test-bed driver; the code is available online [21]. We tested round trip times between two modules connected through the broker using five different scenarios: (1) 10 000 empty message; (2) 1000 short arrays of 10 `double`s each; (3) 100 long arrays of 1000 `double`s each; (4) 1000 short strings of 100 characters each; and (5) 100 long strings of 10 000 characters each.

The communications tests involved only the local machine, which means that network latencies can be assumed to be zero, and the results can be assumed to measure the performance of the message-handling code only. We also attempted to eliminate some other possible confounding variables:

- We performed timing tests under both Linux (Red Hat Linux 7.0, kernel 2.4.0) and Windows (Windows 2000) systems, using identical hardware in both cases (733 Mhz Pentium-III based computer with 384 MB RAM). The test results were comparable and did not reveal a platform-specific advantage of using one OS over the other. Here we present only the results from the Linux-based tests.

- We tested different versions of the Java Development Kit (JDK), specifically Sun's version JDK 1.3 (Java HotSpot Client VM, build 1.3.0rc1-b17, "mixed mode") and IBM's version 1.3 JDK (build 1.3.0, J2RE 1.3.0 IBM build cx130-20010207, JIT enabled), both under the Linux system. We did not find an advantage to using one or the other JVM for either of the

message-passing schemes. (The absolute performances differed when using different JVMs, but the relative performance difference between XML-RPC and SBW-RPC stayed the same.) The results presented in Section A.2.2 were generated using the Sun implementation.

For the XML-RPC portion of these tests, we started with an off-the-shelf solution available on the Internet (version 1.0 beta 4 of the Helma XML-RPC implementation [39]). We tested several XML parser implementations (Ælfred version 1.1 [26], Crimson version 1.1 [2], MinML version 1.0 [40], OpenXML version 1.2 [28], XP version 0.5 [12], and Xerces version 1.3.1 [3]), and found that MinML [40] gave the highest performance on the test suite used here.

Initial tests showed that long arrays of `double`'s took inordinately long to transfer, so we performed one optimization: we changed the XML-RPC implementation to encode arrays of `double`'s as base64-encoded character arrays. (Strictly speaking, this is not in the spirit of XML-RPC, but it is also not directly forbidden.) The test results here are based on this modified implementation.

A.2.2 Results of Timing Experiment

The timing results are presented in Table 1. The table shows that in terms of absolute performance, XML-RPC message exchanges for short messages took 1.3–1.6 ms mean time, whereas the SBW-RPC solution took 0.19–0.28 ms mean time. This represents a factor of 5 to 8 in difference. We expect short messages to be the most relevant case for most applications. The time difference is similar for the case of empty messages; there again, the XML-RPC approach took approximately ten times as long to exchange messages as the SBW-RPC method. For long messages, exchange times were similar for both methods.

Test	Mean Run Times (ms) XML-RPC	SBW-RPC
short array of `double`'s	1.6	0.19
short character string	1.3	0.28
long array of `double`'s	11.3	8.6
long character string	8.9	8.2
empty message	1.2	0.10

Table 1: Mean run times for each case of the timing experiments.

A.3 Analysis and Discussion

The results in the previous section imply that message exchanges using the XML-RPC approach can take up to ten times as long as in SBW-RPC. The results indicate that SBW-RPC is a higher-performance scheme than XML-RPC.

It is important to consider these timing results in the context of expected run times in an application. One situation that will demand high performance is optimization involving separate packages, in which the objective function requires running a simulation (e.g., using an ODE-based simulator such as Jarnac [30]) under the control of the optimizer (e.g.,

Gepasi [25]). This is admittedly not an ideal configuration; implementing the objective function inside the optimizer module would be more efficient, but sometimes this is not feasible. This scenario will require repeated message exchanges back and forth between separate modules. We attempted to estimate how long a typical simulation might take by examining a few example Jarnac simulations and then making some rough estimates for other cases. The results are shown in Table 2.

Case	Execution times
Run to steady-state	
Small model	1.5–5 ms
Large model (estimated)	15–50 ms
Time-course simulation	
Small model	10–30 ms
Large model (estimated)	40–70 ms

Table 2: Approximate and estimated run times for simulations using Jarnac, based on a small sample problem involving a two-parameter model with three species and non-trivial kinetics. The estimates for a "big model" are educated guesses made by Jarnac's author.

The estimates in Table 2 show that the run time for reaching steady-state in a model (the most likely situation used in an optimization problem) is 1.5 ms *at minimum*. Let us assume that actual ranges will be closer to the mean value of the extremes: so, instead of 1.5–5 ms for a small model and 15–50 ms for a large model, we take $(1.5 + 50)/2 \approx 30$ ms (rounded down) as the average time to compute a result for a realistic model in a system such as Jarnac. If message round-trip times are on the order of 1 ms, as in the XML-RPC tests for short messages, it implies that the total time spent on each computational cycle will be $30+1 = 31$ ms. A message time of 1 ms is equal to $1/31$ of this total, or 3.2%. On the other hand, the SBW-RPC case takes approximately 0.1 ms, which implies that the total time spent on each computational cycle will be $30+0.1 = 30.1$ ms. A message time of 0.1 ms, then, is equal to $0.1/30.1$ of this total, or 0.33%. Note, however, that although the difference in percentages of time spent in communications appears large (3.2% versus 0.33%), the difference in absolute times (31 ms versus 30.1 ms) is small: it is 2.9%. We conclude from these results that, although data transfers in XML-RPC are slower than in SBW-RPC, the impact on real applications would not be significant enough to justify making a choice solely on this basis.

A.3.1 Other Factors

A few other factors could, however, affect the choice of one messaging framework versus the other. One is that XML-RPC uses standards such as HTTP and XML, and therefore has a certain attractiveness that a novel, homegrown approach such as SBW-RPC lacks. However, it is important to note that the message format and protocol are hidden behind programming libraries in SBW, and few developers will care about their actual implementation.

A second selling point for XML-RPC is that, because it uses HTTP as its transport protocol, it is in theory able to cross

most network firewall without special requirements. Crossing a firewall using the SBW-RPC method would require the firewall to allow network traffic on a nonstandard port. Most site administrators, especially in corporate network environments, are reluctant to introduces special-case holes in their network security systems.

Unfortunately, it turns out that for purposes of SBW, using XML-RPC would not make it easier to cross network firewalls. SBW requires bidirectional communications between modules. One module may call on another (via the broker as intermediary), and two modules may exchange information with each other. In all cases, any module can serve as the initiator. One of our realizations during the testing process was that XML-RPC does not support bidirectional connections. The problem lies in the HTTP protocol, which is oriented towards client-server applications where an agent initiates a connection to a server listening on a designated TCP/IP port. The implication of using XML-RPC for SBW is that if modules A and B needed to invoke operations on each other simultaneously, module A would have to initiate a connection to B and B *would also* have to initiate a *separate* connection to module A. Although this would itself not be a problem, it has two important implications: every module would need to be assigned a unique TCP/IP port on a given machine, so that it could listen for requests directed at it; and the firewall transparency implied by using HTTP would be lost, because modules would have to use ports other than the standard HTTP port—communications could not all take place through a single connection through a firewall as in normal HTTP. It appears the only work-around to this would be to violate the HTTP protocol and somehow tunnel a bidirectional data stream over a single HTTP connection. However, such an implementation would not adhere to the XML-RPC specification and would not be compatible with existing XML-RPC implementations. Thus, XML-RPC would offer no advantages over SBW-RPC in terms of firewall transparency.

There is one factor that does have a substantial impact on the choice of messaging framework used for SBW: support for required data types. The data types made available in the SBW-RPC implementation were chosen (1) to support the types of data that we expected would be needed by systems biology simulators, and (2) to provide a reasonable balance between flexibility and conciseness. We knew that data types such as IEEE floating point would be important for simulation software. It is therefore important to note that XML-RPC does not specify the use of IEEE floating point numbers: XML-RPC's floating point type is `double`, whatever that may be in a given programming language, and when written out in XML, the double is expressed as a sequence of ASCII characters (e.g., `"234.255"`). As a consequence, there is no formalized way of expressing overflow and NaN; moreover, representing numbers in character string form can lead to a loss of numerical precision. Thus, using XML-RPC as the basis of the SBW communications framework would require that we add mechanisms to support IEEE floating-point quantities (perhaps by manually encoding numbers using byte arrays). This would likely have no impact on performance, but it would call into question the logic of using XML-RPC when it does not support the basic data types needed by our application.

A.4 Conclusions

In summary, although our timing tests indicate that SBW-RPC performs better than XML-RPC, the impact of the difference on real applications is likely to be negligible. Therefore, a choice between the two schemes cannot be made on this basis. Several other factors also do not present a clear argument in favor of XML-RPC. However, the lack of support for key data types in XML-RPC *does* present a clear argument in favor of SBW-RPC. For these reasons, we chose SBW-RPC for use in SBW.

B. REFERENCES

[1] G. Allen, W. Benger, T. Goodale, H.-C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf. The Cactus Code: A problem solving environment for the Grid. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC9)*. IEEE Computer Society, 2000.

[2] Apache Software Foundation. Crimson version 1.1. Available via the World Wide Web at `http://xml.apache.org/crimson/index.html`, 2001.

[3] Apache Software Foundation. Xerces version 1.3.1. Available via the World Wide Web at `http://xml.apache.org/xerces-j/index.html`, 2001.

[4] A. P. Arkin. *Simulac* and *Deduce*. Available via the World Wide Web at `http://gobi.lbl.gov/~aparkin/Stuff/Software.html`, 2001.

[5] K. Arnold, editor. *The Jini Specification, 2nd Edition*. Addison-Wesley Publishing Company, 2000.

[6] N. Borenstein and N. Freed. MIME (Multipurpose Internet Mail Extensions). Internet Request for Comments 1341, Bellcore, 1992. Available via the World Wide Web at `http://www.faqs.org/rfcs/rfc1341.html`.

[7] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1: W3C note 08 May 2000. Available via the World Wide Web at `http://www.w3.org/TR/SOAP/`, 2000.

[8] R. Bramley, K. Chiu, S. Diwan, D. Cannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri. A component based services architecture for building distributed applications. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing, Pittsburgh, PA, Aug. 2000*, 2000.

[9] D. Bray, C. Firth, N. Le Novère, and T. Shimizu. *StochSim*. Available via the World Wide Web at `http://www.zoo.cam.ac.uk/comp-cell/StochSim.html`, 2001.

[10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, , and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[11] H. Casanova and J. Dongarra. NetSolve: A network-enabled server for solving computational

science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997.

[12] J. Clark. XP version 0.5. Available via the World Wide Web at `ftp://ftp.jclark.com/pub/xml/`, 1998.

[13] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1999.

[14] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 1996.

[15] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, MA, 1994.

[16] M. Gibson and J. Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *J. Phys. Chem. A*, 104:1876–1889, 2000.

[17] M. Ginkel, A. Kremling, F. Tränkle, E. D. Gilles, and M. Zeitz. Application of the process modeling tool ProMoT to the modeling of metabolic networks. In I. Troch and F. Breitenecker, editors, *Proceedings of the 3rd MATHMOD*, pages 525–528, 2000.

[18] N. H. Goddard, M. Hucka, F. Howell, H. Cornelis, K. Shankar, and D. Beeman. Towards NeuroML: Model description methods for collaborative modelling in neuroscience. *Philosophical Transactions of the Royal Society*, 2001. In press.

[19] I. Goryanin, T. C. Hodgman, and E. Selkov. Mathematical simulation and analysis of cellular metabolism and regulation. *Bioinformatics*, 15(9):749–758, 1999.

[20] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1999.

[21] M. Hucka and A. Finney. Source code for XML-RPC versus SBW-RPC comparison. Available via the World Wide Web at `ftp://ftp.cds.caltech.edu/pub/caltech-erato/message-comparison`, 2001.

[22] M. Hucka, H. M. Sauro, A. Finney, H. Bolouri, J. Doyle, and H. Kitano. The ERATO Systems Biology Workbench: An integrated environment for multiscale and multitheoretic simulations in systems biology. In H. Kitano, editor, *Foundations of Systems Biology*, chapter 6. 2001. In press.

[23] M. Hughes, M. Shoffner, and D. Hamner. *Java Network Programming*. Manning Publications Co., 1999.

[24] B. Janssen, M. Spreitzer, D. Larner, and C. Jacobi. ILU 2.0beta1 reference manual. Available via the World Wide Web at `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`, 1999.

[25] P. Mendes. Biochemistry by numbers: Simulation of biochemical pathways with Gepasi 3. *Trends in Biochemical Sciences*, 22:361–363, 1997.

[26] Microstar Software Ltd. Ælfred version 1.1. Available via the World Wide Web at `http://www.microstar.com/XML/`, 1998.

[27] Open Software Foundation. *OSF DCE Application Development Guide*. Prentice-Hall, 1993.

[28] OpenXML.org. OpenXML version 1.2. Available via the World Wide Web at `http://www.openxml.org`, 2001.

[29] M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H. Takagi. Ninf: A network based information library for a global world-wide computing infrastructure. In *Proceedings of HPCN'97 (LNCS-1225)*, pages 491–502, 1997.

[30] H. M. Sauro. Jarnac: A system for interactive metabolic analysis. In J.-H. S. Hofmeyr, J. M. Rohwer, and J. L. Snoep, editors, *Animating the Cellular Map: Proceedings of the 9th International Meeting on BioThermoKinetics*. Stellenbosch University Press, 2000.

[31] H. S. Sauro. JDesigner: A simple biochemical network designer.

[32] J. Schaff, B. Slepchenko, and L. M. Loew. Physiological modeling with the Virtual Cell framework. In M. Johnson and L. Brand, editors, *Methods in Enzymology*, volume 321, pages 1–23. Academic Press, San Diego, 2000.

[33] A. Siepel, A. Farmer, A. Tolopko, M. Zhuang, P. Mendes, W. Beavis, and B. Sobral. ISYS: A decentralized, component-based approach to the integration of heterogeneous bioinformatics resources. *Bioinformatics*, 17(1):83–94, 2001.

[34] Sun Microsystems. XDR: External data representation standard (RFC 1014). Internet Request for Comments 1014, Sun Microsystems, Inc., 1987. Available via the World Wide Web at `http://www.faqs.org/rfcs/rfc1014.html`.

[35] Sun Microsystems. Java™ object serialization specification. Technical report, Sun Microsystems, Inc., 1998. Available via the World Wide Web at `http://java.sun.com/products/jdk/1.2/docs/guide/serialization/`.

[36] D. Sweet. *KDE 2.0 Development*. Sams, 2001.

[37] M. Tomita, K. Hashimoto, K. Takahashi, T. Shimizu, Y. Matsuzaki, F. Miyoshi, K. Saito, S. Tanida, K. Yugi, J. C. Venter, and C. Hutchison. E-Cell: Software environment for whole cell simulation. *Bioinformatics*, 15(1):72–84, 1999.

[38] S. Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communication*, Feb. 1997.

[39] H. Wallnoefer. Helma XML-RPC library for Java, version 1.0 beta 4. Available via the World Wide Web at `http://xmlrpc.helma.org/`, 2001.

[40] J. Wilson. MinML version 1.0. Available via the World Wide Web at `http://www.wilson.co.uk/`, 2001.

[41] D. Winer. XML-RPC specification. Available via the World Wide Web at `http://www.xmlrpc.com/spec/`, 2001.