Building IAMC: A Layered Approach

Weidong Liao(Presenter)[*] **Paul S. Wang**[†]
Institute for Computational Mathematics
Department of Mathematics & Computer Science
Kent State University
Kent, Ohio 44242, U.S.A.
Telephone: (330) 672-4004 ext. 111
Fax: (330) 672-7824
`{wliao, pwang}@mcs.kent.edu`

Abstract

Internet Accessible Mathematical Computation (IAMC) is a distributed system to make mathematical computation easily and widely available on the Web/Internet. The architecture and implementation of a framework for building IAMC systems are presented. Protocol layers, allowing any well-defined encodings for mathematical data, connect the IAMC client (Icl) and server (Isv). An *external engine interface* (EEI) specification makes connecting existing compute engines straight-forward. A Java-based IAMC framework prototype has been implemented including Icl, Isv, protocol layers, and the Java *Compute Engine Connectivity* (JCEC) API as a realization of the EEI.

Keywords: Internet, Mathematical Computation, IAMC, Protocol, Java

1 Introduction

Internet Accessible Mathematical Computation (IAMC) [9] is a research project at the Institute of Computational Mathematics (ICM) at Kent State University. The goal of IAMC is to make all kinds of mathematical computations easily accessible on the Web and the Internet. The topic is a focal point of discussion at the 1999 *IAMC Workshop* part of the *International Symposium on Symbolic and Algebraic Computation* (ISSAC'99), held late July in Vancouver, Canada. Making mathematical information and computation available in the new communication age is an important and active area of research and development. For more background and related activities, the reader is referred to the online Proceedings of the IAMC'99 Workshop [1], the IAMC homepage[2], and the Workshop on *The Future of Mathematical Communication* (FMC Dec. 1999)[3].

[*]Work reported herein has been supported in part by the Ohio Board of Regents Computer Science Enhancement Funds

[†]Work reported herein has been supported in part by the National Science Foundation under Grant CCR-9721343

[1]`http://horse.mcs.kent.edu/icm/research/iamc99proceedings.html`

[2]`http://horse.mcs.kent.edu/icm/research/iamc`

[3]`http://www.msri.org/activities/events/9900/fmc99/`

Protocols and frameworks help make an increasing number of new services accessible on the Web and Internet. The Java Database Connectivity, Enterprise JavaBeans, and Servlets and the Microsoft ODBC, DCOM and Active Server Page are examples. The same approach should work well for mathematical computation. The IAMC project designed a series of protocols and specifications to provide a framework for making many kinds of mathematical computations accessible on the Internet. A layered protocol model is used. The *Mathematical Computation Protocol* (MCP) sits on top of a reliable transport layer and allows the use of any well-defined mathematical data encodings. A session control layer complements the MCP. The *external engine interface* specification provides a systematic way to adopt existing mathematical computation engines to work with IAMC. The design and specification of these protocols are still evolving. We are also building a prototype IAMC framework to test and refine the protocol model and specifications.

The Java implementation of the IAMC framework prototype is the topic here. We will first review the overall IAMC architecture and the protocol layers. The prototype structure as well as the object-oriented design and Java implementation of the client (Icl) and the server (Isv) will then be presented in some detail. Modeling after the JDBC, we also present an EEI specification and implementation that we call JCEC (the Java Compute Engine Connection).

2 IAMC Overview

IAMC is a distributed system to make mathematical computing easily accessible and usable on the Internet [9]. IAMC consists of the following components:

1. IAMC client (Icl) — The end-user agent for accessing services provided by any IAMC server. The Icl could be an interactive agent with a graphical user interface, a command-based text-only agent, a Web-based facility using HTML forms or Java applets, or a email-based agent.

2. IAMC server (Isv) — The server providing computational services through the IAMC protocol layers. An Isv may or may not employ an external compute engine to perform mathematical computations.

3. Protocol layers — A series of protocols used for connecting IAMC clients and servers. They are the IAMC Application Layer, the IAMC Session Control Protocol (ISCP) layer, and the Mathematical Computation Protocol (MCP) layer.

4. Mathematical Data Encoding — Standard or user-defined mathematical data encoding formats. Standard formats allow heterogeneous mathematical programs to interoperate with one another. Other formats that a particular pair of Icl and Isv can handle can also be used. The IAMC framework allows plug-in format converters, on the client and the server side, to support additional formats.

5. The External Engine Interface (EEI) — A specification and API implementation for binding existing compute engines to IAMC servers.

Figure 1 shows the overall IAMC architecture.

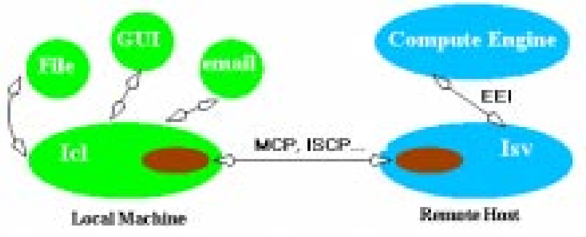


Figure 1: IAMC Architecture

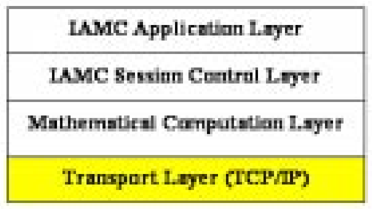


Figure 2: IAMC Protocol Model

2.1 IAMC Protocols

The IAMC protocol layers are shown in Figure 2. The application is the Icl on the client side and the Isv on the server side. The Icl is the user agent that uses the IAMC protocol to access a given Isv.

The IAMC Session Control layer provides the IAMC Session Control protocol (ISCP) for callback and asynchronous service invocation, for interrupting server-side computations, and for IAMC URL resolution.

The MCP layer (Mathematical Computation Protocol) supports interactive mathematical computation based on a persistent connection. MCP uses HTTP-style request/response transactions and defines message formats to be transferred between the Icl and Isv. Major elements of MCP have been specified and implemented. Building the IAMC prototype allows us to test and refine MCP.

2.2 EEI Specification

External Engine Interface (EEI) Specification is given as a guideline to design compute engine drivers that can accept an EEI call, translate it into native compute engine commands, and fetch the result(s). The EEI specification was carefully designed after studying several existing symbolic and numeric systems. The following properties were noticed:

- Compute engines may execute some initialization commands before it begins to do real computation. Usually, those initialization commands are used to set up the input mode, the output mode, the exit command, etc.
- Compute engines may execute in several different modes such as regular mode and debug mode.
- Compute engines send a prompt when they are ready for input. Different compute engines have different prompt. This prompt may give some information about the status of the engine.
- Compute engines take one command at a time. Each command has a terminating character.
- Compute engines return a result or error message to the user. Results may contain text, mathematical expressions, and graphics. There should be some methods to identify the beginning and end of each result.

- During a particular computation, a compute engine may ask for extra information from the user in order to complete this computation.
- Compute engines may support several types of user-generated interrupts.
- Compute engines can support history replacement.
- Help information and documentation can come from the engine or by using an external program.

The Isv and the compute engine can run as separate processes on the same machine and perform interprocess communication, or reside in the same local area network and perform standard communications. Once the Isv receives a session request from an Icl, the Isv looks up and contacts a compute engine driver. The engine driver in turn will pre-start the compute engine, do some initialization steps, decide what kind of expressions the compute engine can accept, and figure out how to provide help and documentation services. The engine driver finds necessary information in a property file, which corresponds to the configuration of a compute engine.

In addition to the command/result interface, the EEI also specifies a help/documentation interface to the external engine. It is required to specify how EEI can get help/documentation/template service from the compute engine. If it is difficult to get this service from the compute engine, some other facilities need to be provided. The common practice is to specify a URL or filename. All these settings are stored in the property file.

From developers' point of view, the EEI specification is a call-level programming interface and the software package implementing this call-level interface is the EEI driver. The EEI is language independent; it can be specified further in any programming language. The EEI specification for Java programming language, called Java Compute Engine Connectivity (JCEC), has been defined and a sample JCEC driver has been developed as an essential part of our IAMC prototype. The JCEC API is presented in Section 6.

3 IAMC Prototype Structure

Implemented in Java, the prototype IAMC framework presented here has a three-tier structure, as shown in Figure 3. Tier one is the Icl. *Dragonfly* is our Java-based Icl prototype. Dragonfly supports interactive use both through a GUI and through a text-only command mode. Access via email is also possible.

Starfish is the prototype Isv for the second tier. *Starfish* is responsible for providing remote users with particular mathematical computation capabilities. It may perform the computation internally or use external compute engines. The server obtains the results, encoding and delivering them to the Icl. It is also possible for Starfish to perform task division and load-balancing for advanced distributed/parallel computations.

If an external compute engine is used, then it becomes the third tier. Through EEI drivers, existing computational packages, such as Maxima and Maple, can be seamlessly integrated into the IAMC framework. Starfish controls a slightly modified version of the MAXIMA symbolic computation system (Common Lisp based) and it communicates with MAXIMA through a MAXIMA JCEC driver.

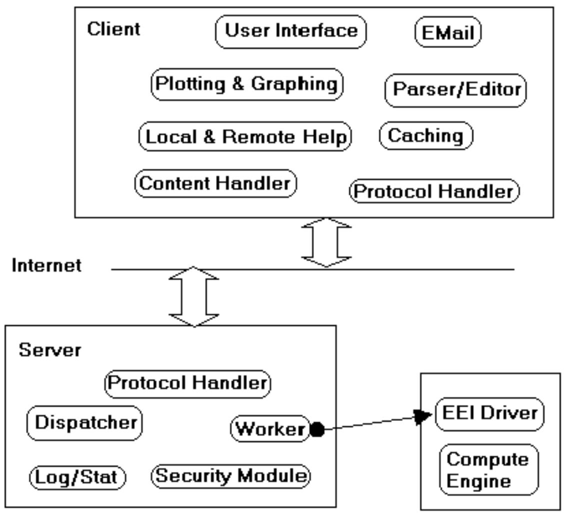


Figure 3: IAMC System Components

4 The Dragonfly Client

The main components and the corresponding Java class packages for Dragonfly are listed as follows.

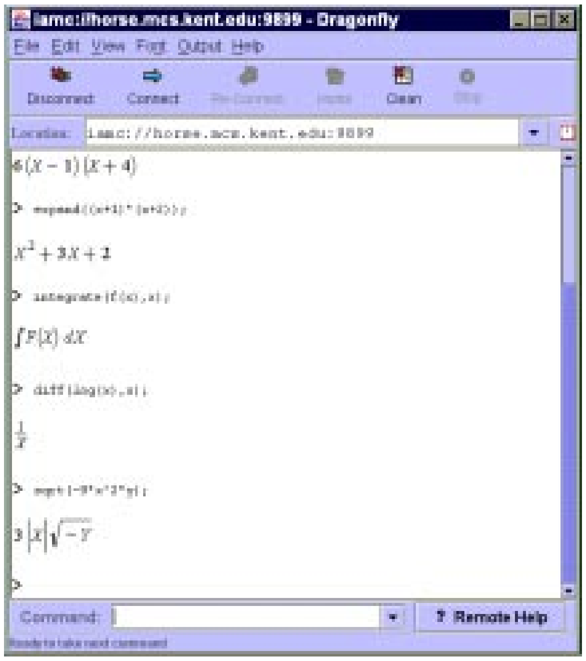
- *org.icm.IAMC.icl* - The Overall Control Classes for Dragonfly
- *org.icm.IAMC.icl.ui* - User Interface Package
- *org.icm.IAMC.icl.render* - Mathematical Expression Rendering Package
- *org.icm.IAMC.icl.plot* - 2-Dimensional and 3-Dimensional Plotting
- *org.icm.IAMC.icl.parser* - Parser for mathematical expression
- *org.icm.IAMC.icl.help* - Help library
- *org.icm.IAMC.icl.util* - Utilities
- *org.icm.IAMC.icl.MCP* - MCP Protocol Library
- *org.icm.IAMC.icl.ISCP* - IAMC Session Control Protocol Library

Java 2 platform from Javasoft and WebEQ [12] from Geometry Technologies are main development environment of Dragonfly. In particular, Java Swing from Java 2 package is employed to implement the user interface and the HTML-based help facility, and Java 2D is used to plot 2-dimensional and 3-dimensional mathematical graphics. Finally, WebEQ package is leveraged to render mathematical expressions in 2-dimensional format.

Figure 4 illustrates Dragonfly main user interface. Computation commands are typed in the list box **Command:** , and those commands will be echoed in the main window simultaneously.

Press **Enter** key, the current command in **Command:** box will be sent to the Isv, and the returned result will be shown in the main window in 2-dimensional format.

The button labelled **Remote Help** can be employed to get help about computational commands and their usage from the Isv that Dragonfly is currently connecting to. Click on this button, Dragonfly will contact the Isv and issue a request about the current content in **Command:** box , fetch the context-sensitive help information, and display this information in a separate **Help** window. If the **Command:** box is empty, a general description about the Isv will be displayed.


iamc://horse.mcs.kent.edu:9899 - Dragonfly
File Edit View Font Output Help
Disconnect
Connect
Re-Connect
Home
Clean
Stop
Location:
iamc://horse.mcs.kent.edu:9899
Command:
? Remote Help
Ready to take next command

```
6(X − 1)(X + 4)

> expand((x+1)*(x+2));

X^2 + 3X + 2

> integrate(f(x),x);

∫F(X) dX

> diff(log(x),x);

1/X

> sqrt(-9*x^2*y);

3|X|√(−Y)

>
```

Figure 4: *M*ain User Interface

Dragonfly supports 2-dimensional and 3-dimensional plotting. To plot a 2-dimensional or 3-dimensional math graph, type the plotting commands in the **Command:** box as usual. The plotting command and related parameters are sent to the Isv, which in turn delivers the plotting commands to a compute engine. Once the compute engine finishes, the Isv will encode graphics data in a format that is acceptable for Dragonfly. (The acceptable data formats are set during the MCP connection initialization time [11]. When Dragonfly receives the encoded graphics data, it decodes them and renders the math graph in a separate *Grapher* window. Figure 5 shows the 2-D and 3-D plotting window.

5 The Starfish Server

The functional requirements and features for an Isv can be summarized as follows:

- read the configuration data,
- wait for connection request,

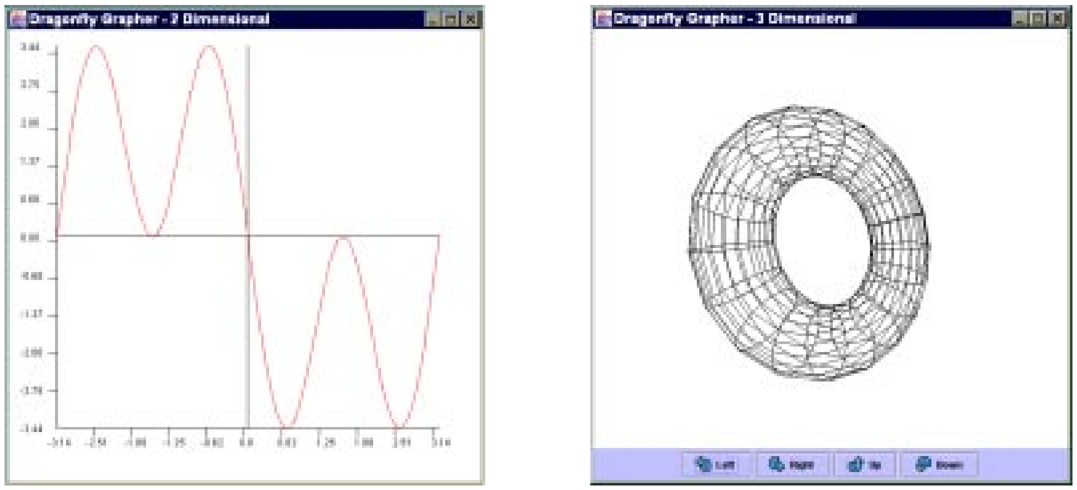

Dragonfly Grapher - 2 Dimensional
Dragonfly Grapher - 3 Dimensional
Left
Right
Up
Down

Figure 5: *2*-d and 3-dimensional Plotting

- create a session,
- start the compute engine,
- forward computational requests to compute engines,
- process information requests such as help, documentation and command template,
- deliver results to the client over the current session channel, either synchronously or asynchronously,
- When required, perform callback (dialog) requests over the current session channel,
- destroy the session when a request has been served completely,
- close the connection when the client wants to exit,
- support multi-threading so that multiple clients can be served concurrently,
- dynamically load computational services so that multiple engines can be attached to each server, and those engines can be started separately and upon request from the Icl,
- perform log operation and authentication.

We have developed a Java based multi-threaded Isv prototype called *Starfish*. Besides the above basic requirements, *Starfish* also has a graphical user interface for users to configure it and monitor its activities.

5.1 Mulit-threading

In *Starfish*, threads are spawned to handle multiple clients and multiple sessions. Two threading strategies are used. For "multiple clients", the thread-pre-request strategy is employed. For "multiple sessions", the thread pool model is chosen.

In the thread-per-request strategy, an *acceptor* thread spawns a new thread for each new coming request. The newly spawned thread is called the *connector* thread that is used to handle a specific connection between an Icl and Isv. Once the *connector* is created, the *acceptor* thread continues waiting for new connections.

The *connector* thread is responsible to handle multiple sessions. Since the thread pool model is employed, this *connector* thread would pre-spawn a group of threads. When a session request arrives, the *connector* selects a thread from the pool to handle this session.

The reason to choose thread-per-request strategy for connector threads is that the connection between an Icl and Isv usually lasts for relatively long time and this strategy is very adequate for long-duration request. In contrast, sessions are usually short-term entities so that the thread pool model can eliminate the overhead of creating a new thread for each session and thus reduce the response time.

5.2 Dynamic Service Binding

To dynamically bind compute engines into *Starfish*, a property file is used to specify the list of accessible engines and their properties. Based on the property file, *Starfish* can locate suitable JCEC drivers and load them as necessary. Currently, we have developed JCEC driver for Maxima. The development of JCEC driver for Maple is undergoing.

In addition, *Starfish* leverages Java Reflection API to load and instantiate a named engine classes dynamically. To a simplest extent, this flexibility is achieved by using *Class* class in Reflection API whose instances represent classes and interfaces in a running Java application. The following code section shows how to do it:

```
String engineName;
Class engineClass = Class.forName(engineName); //dynamically load
Engine engine = (Engine)engineClass.newInstance();
```

6 Java API for EEI

Java-based External Engine Interface is called Java Compute Engine Connectivity (JCEC). It is a Java API specification used to declare compute engine access pattern. A JCEC driver can be thought of as a Java compute engine wrapper. It encapsulates details such as connecting to a compute engine, submitting computational tasks, fetching results, getting help or command templates, etc.

The JCEC is designed after studying Java language features and the scenario on how the Isv accesses compute engines. Since the compute engine is pre-started when the Isv receives a connection request from the client, this scenario only deals with the procedure to submit command to compute engines and fetch results. The steps involved in the scenario can be described as follows:

1. Connect to the compute engine.
2. Prepare a Command object.
3. Execute the command.
4. Retrieve the result.

5. Release the Command object
6. Release the connection.

From the above scenario, the following Java classes and interfaces can be outlined:

1. **Connection**
   A *Connection* object represents a session with a specific compute engine. It provides a context for executing computational commands and processing their results. The methods in a *Connection* object include:
   - *public Command createCommand()* - create a command context.
   - *public EngineMetaInfo getMetaInfo()* - retrieve the meta information about the engine connected.
   - *public Help getHelp()* - get help information about the engine connected.
   - *public Template getTemplate()* - get command template about a specific statement.
   - *public void close()* - close the connection.
2. **Command**
   A *Command* object is used for executing a computational command and obtaining the results produced by it. Optionally, it can also be used to execute a batch of commands and fetch result set in one time. The following methods are defined in this object:
   - *public void execute(String command)* - execute the given command.
   - *public Result getResult()* - retrieve the result.
   - *public void cancel()* - cancel the execution if both the JCEC driver and compute engine support the Abort operation.
   - *public Connection getConnection()* - get Connection object that produced this Command object.
3. **Result**
   The *Result* object provides methods that let you access the result of a command execution. It methods include:
   - *public int getType()* - return the type of this Result object.
   - *public String getString()* - fetch the value as a string.
   - *public Integer getInteger()* - fetch the value as an Integer.
   - *public Float getFloat()* - fetch the value as a Float.
   - *public Double getDouble()* - fetch the value as a Double.
   - *public InputStream getAsciiStream()* - fetch the result as an ASCII stream.
   - *public InputStream getBinaryStream()* - fetch the result as a Binary stream.
   - *public InputStream getDoubleStream()* - fetch the result as an Double stream. This is useful for mathematical graphic data.

Besides the above classes, there are also some supporting interfaces and classes that are essential to a JCEC driver. The most important are **Driver** interface and **DriverManager** class. The **Driver** is an interface that every JCEC driver must implement. Once a driver is loaded, it creates an instance of itself and then registers it with the DriverManager.

The **DriverManager** class manages JCEC drivers and dispenses **Connection** objects. As part of initialization, the class will locate and load the JCEC drivers the user specifies in the jcec.drivers property. A newly loaded driver class calls *registerDriver()* to make itself known to the **DriverManager**, and *deregisterDriver()* to remove itself from the driver list.

Currently, compute engines are located by their identifiers. This is sufficient because compute engines are assumed to reside on the same host as the corresponding Isv. This naming convention could be extended easily by employing a compute engine URL in the following naming scheme:

Jcec:⟨ domain name ⟩ [:port]/ ⟨ compute engine identifiers ⟩

The *getDriver()* method in **DriverManager** class is used to locate a driver that understands the given identifier. The *getConnection()* method is used to establish a connection to the given identifier.

The following class description gives more detailed information about the **Driver** interface and **DriverManager** class:

```
public interface Driver {
    public abstract Connection connect(String engineName);
    public abstract String [] getPropertyInfo(String engineName);
    public abstract boolean isJcecCompliant();
    public abstract int getVersion();
}
public class DriverManager {
    public static synchronized Connection getConnection(String engineName);
    public static Driver getDriverByName(String engineName);
    public static Enumeration getDrivers();
    public static synchronized void registerDriver(Driver driver);
    public static void deregisterDriver(Driver driver);
}
```

7 Future Work

The IAMC encompasses a wide range of networking technologies and mathematical services. The goal for our layered framework is to provide a finer modularized architecture and thus facilitate development of IAMC compatible systems. Continuing work on this framework includes fully specifications of layered protocol model, refining our IAMC Server and Client prototypes, building more EEI engine drivers that allow more demonstration IAMC services, and investigating performance results for our prototyping system.

References

[1] ABBOTT, J., DIAZ., A., AND SUTOR, R. S. *Report on OpenMath.* ACM SIGSAM Bulletin (Mar. 1996), 21-24.

[2] BORENSTEIN, N., AND FREED, N. *MIME: Mechanism for Specifying and Describing the Format of Internet Message Bodies.* The Network Working Group RFC-1521, Sept. 1993.

[3] DOLEH, Y., AND WANG, P. S. *SUI: A System Independent User Interface for an Integrated Scientific Computing Environment.* In Proc. ISSAC 90 (Aug. 1990), Addison-Wesley (ISBN 0-201-54892-5), pp. 88-95.

[4] FATEMAN, R. J. *Network Servers for Symbolic Mathematics. In Proc. ISSAC'97 (1997),* ACM Press, pp. 249-256.

[5] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., AND BERNERS-LEE, T. Hypertext Transfer Protocol - HTTP/1.1, Jan. 1997.

[6] GRAY, S., KAJLER, N., AND WANG, P. *MP: A Protocol for Efficient Exchange of Mathematical Expressions.* In Proc. ISSAC'94 (1994), ACM Press, pp. 330-335.

[7] GRAY, S., KAJLER, N., AND WANG, P. *Design and Implementation of MP, A Protocol for Efficient Exchange of Mathematical Expressions.* Journal of Symbolic Computation 25 (Feb. 1998), 213-238.

[8] ION, P., MINER, R., BUSWELL, S., S. DEVITT, A. D., POPPELIER, N., SMITH, B., SOIFFER, N., SUTOR, R., AND WATT,S. *Mathematical Markup Language (MathML) 1.0 Specification.* (www.w3.org/TR/1998/REC-MathML-19980407), Apr. 1998.

[9] WANG, P. S. *Design and Protocol for Internet Accessible Mathematical Computation.* In Proc. ISSAC'99 (1999), ACM Press, pp. 291-298.

[10] WANG, P. S. *Internet Accessible Mathematical Computation.* In Proc. 3rd Asian Symposium on Computer Mathematical (ASCM'98) (Aug. 1998), pp. 1-13.

[11] WANG, P. S., GRAY, S. *Mathematical Computational Protocol Specification.* Initial Draft. Oct., 1999.

[12] WebEQ Home Page. http://www.webeq.com/.