

Experiments with JSP, XML, CORBA and HDF5

Kun Yan

Robert E. McGrath

National Center for Supercomputing Applications

University of Illinois, Urbana-Champaign

August, 2002

Contents

1. Introduction	1
2. Experiment 1: JSP and XML	2
2.1. Introduction	2
2.2. Software Configuration	3
2.3. Experiments with HDF	5
2.4. Results and Discussion	6
3. Experiment 2: HDF5 Java/CORBA Experiments	6
3.1. Introduction	6
3.2. Environment setup	7
3.3. Software Development	7
3.4. Experiments with HDF	8
3.5. Summary and Discussion	10
4. Discussion	12
5. Acknowledgements	13
6. References	13

1. Introduction

This project investigated several new technologies that appear to enable HDF5 files to be accessed by Web-based tools. The overall motivation is a desire to make data from HDF files easily accessible using standard Web technology. This requires support at the server, the client, or both [15].

The basic scenario is “browsing” and HDF5 dataset, preferably using a standard Web browser. This process requires a “conversation with the data” ([15]), discovering the structure and objects in the file, and selecting what data to retrieve (download).

In earlier work, we created a CGI-based server for HDF4 (the SDB [1]), which evolved into DIAL [7] and its descendants. Early versions of the NCSA Java HDF Viewer JHV) featured a client-server architecture which used raw sockets, RMI, or Servlets ([16]). The JHV was designed to work as an application; the Java classes were not used by standard Web browsers.

The current HDF Java products access HDF using the Java Native Interface (JNI) which is used to call the native C libraries from Java. Since Java security usually does not allow an applet to call a C library, HDF can be accessed only by Java applications, not by remote Web browsers. This is an unfortunate limitation.

The JNI technology has other serious limitations and flaws, especially for calling a large, complex library that performs I/O. The Java Virtual Machine (JVM) manages the linking and

loading of the native modules, which has many undocumented and platform specific pitfalls. Also, the Java to C interaction runs entirely within the context of the JVM, which is multi-threaded, and has other undocumented details. For instance, the Java specification requires all C-code called by JNI to be ‘thread-safe’—but the definition of ‘thread safe’ is poorly documented and appears to be JVM specific. As far as can be determined, it is not wise to do disk I/O in a native call—which is a severe restriction on the HDF libraries.

For these reasons, it would be advantageous to have an alternative method to access HDF, one that does not use the JNI, and preferably a method by which Web browsers can access data.

Conceptually, it seems clear that Java is designed to communicate with non-Java code via sockets or network services. This can be done through any of several mechanisms, such as HTTP with CGI or Java Server Platform (JSP), or directly over a socket. The connection might exchange messages in many formats, including XML, HTML, or a network protocol such as ODBC. Most of these mechanisms can work across a network or within a single application (possibly as two processes).

A potential advantage of this decoupling of the client from the server is that it may enable the client to be pure Java (i.e., no JNI) and the HDF access to be done without the JNI. It also offers the possibility of an improved “HDF server”, which could enable users to publish HDF data on the Web.

This project investigated several new technologies that appear to enable HDF5 files to be accessed by Web-based tools. Several experiments were done with Java Server Platform (JSP), XML, and HDF-5 for remote accessing scientific data. This experiment is using Tomcat as Web server and JSP servlet container, converting HDF5 file into XML file and further transforming XML file into HTML file using XSL style sheets.

A second phase of this project investigated using CORBA technology with Java to have HDF5 file access done from a remote and heterogeneous environment. The overall approach is to have data access in a CORBA servant written in C/C++, while browsing and presenting the information in Java. Java can communicate directly with CORBA objects via Java’s RMI. Basically, the Java program sees the CORBA object as a remote Java object, although the object can be written in C or C++.

These experiments demonstrated that these technologies can be used to implement sophisticated Web-based browsing for HDF5 files. The JSP and JavaBean server is particularly easy to customize, which would enable specific user communities to create custom “views” of the same HDF5 dataset. The CORBA server is quite robust and reliable, and would be quite useful as an alternative to the JNI.

A combined system with both CORBA and JSP would be quite powerful and flexible. The CORBA server would provide reliable access to HDF5 for clients written in any language, the JSP server could “wrap” the CORBA with different views and dialogs.

2. Experiment 1: JSP and XML

2.1. Introduction

Experiment 1 investigated remote access to HDF5 datasets using Java Server Platform (JSP) and XML. This experiment used standard open source software: the Apache Web server with Tomcat JSP servlet container [citations], which dynamically generates HTML or XML using

JavaBeans™. A set of JSP pages was created which convert HDF4 to HDF5, and HDF5 to XML, and XML to HTML, which is sent to a standard Web browser.

This experiment built on other work with XML, in which we created an XML DTD for HDF5 [17,18], and tools for converting to and from HDF5 binary files and XML, and also used XSL stylesheet to translate the XML into HTML [19]. We also used the *h4toh5* utility [20], which converts HDF4 files to HDF5.

The first goal of this experiment was to explore the JSP and XML technology, to learn the terminology and techniques, and gain practical experience with the software configuration. The second goal was to construct a demonstration server, which converts HDF5 into HTML or XML.

2.2. Software Configuration

A basic system was set up, and small JSP examples use. From this experience, a simple experiment was set up. Figure 1 shows a functional diagram of the how JSP works in this experiment [2, 3, 4].

When the client request a JSP page, the file's extension, `.jsp`, tells the server a special handling needed. The request is forwarded to Tomcat from Apache. The special handling involves four steps [5]:

1. The JSP engine parses the page and creates a Java source file.
2. It then compiles the source file into a `class` file. The class file is a servlet, and from this point on, the servlet engine handles the `class` file in the same manner as all other servlets.
3. The servlet engine loads the servlet class for execution.
4. The servlet executes and streams back the results to the requestor.

Figure 2 shows the flow of a client's request. (Adapted from Eden and Ludke [5]).

The first step was to set up the required environment: the Apache web server (using Apache), the JSP/Servlet container (Tomcat), and the communicator between Apache and Tomcat, `mod_jk`. For the experiment, the HDF5 library, XML DTD, and some tools are also. Table 1 provides the information about where to find the source/binary code and the instruction of configuration and installation.

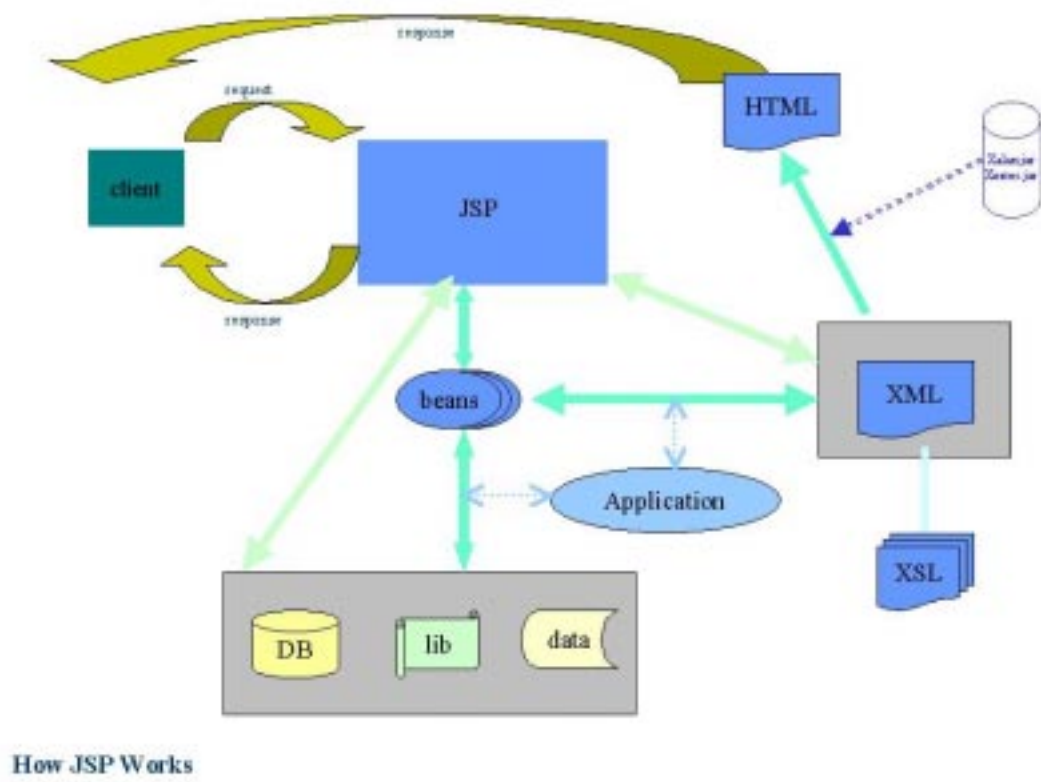


Figure 1. How JSP works in this experiment

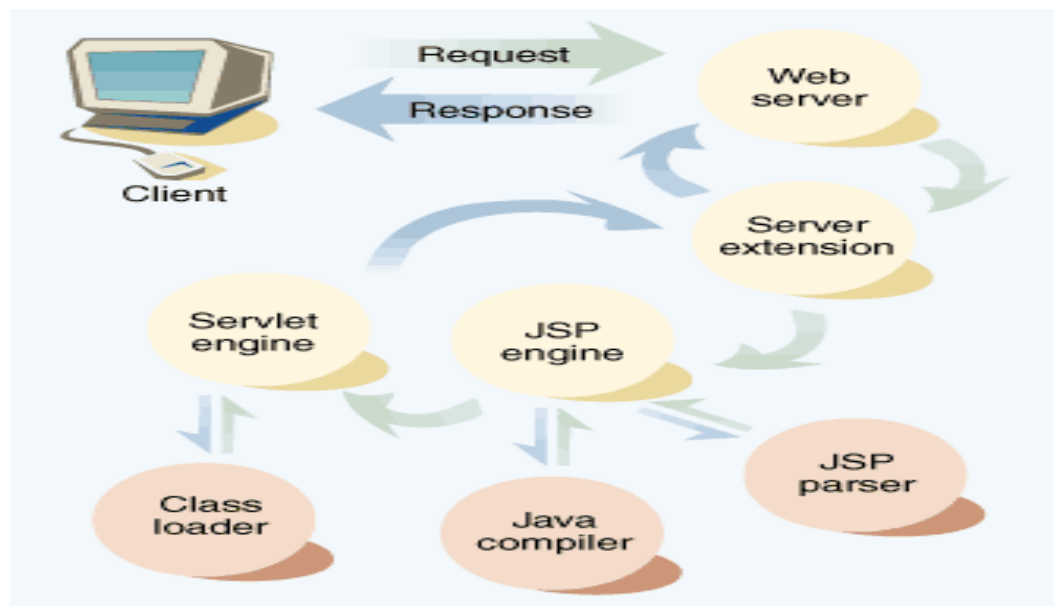


Table 1 *Apache, Tomcat, and other software used in this experiment*

	Download
Apache Web Server	[21]
Tomcat 3.3	[22]
mod_jk	[22]
hdf5 tools	[14, 20]
DTD for hdf (5.1.4)	[23]

The “mod_jk” is a Tomcat-Apache plugin that handles the communication between Tomcat and Apache. We need to reconfigure Tomcat and Apache to make them working with mod_jk. For Apache 1.3.20 and Tomcat 3.3, the reconfiguration is much simpler than with earlier versions. (See [22])

The Tomcat server can be configured as a stand alone Web server, but in this experiment it was configured as a backend to the regular Apache Web server. This is the typical use of Tomcat, which is not as fast as Apache when it serves static pages. The Apache server handles the static content, such as images and HTML documents, and forwards all requests for dynamic content to Tomcat [2].

2.3. Experiments with HDF

This experiment was implemented by four JSP pages and three JavaBean classes. The pages of HDF5, XML, and HTML file lists are generated by the same JSP file and JavaBeans with different parameters passing to corresponding JSP pages. The Java beans access HDF5 using the Java Native Interface (JNI) to call the native HDF5 library.

The test data included a sample of HDF5 files and some real NASA data files. Those files were converted from HDF4 to HDF5 by *h4toh5* utility. Table 2 lists the NASA test data.

Table 2. *NASA datasets that were be converted to HDF5.*

HDF5 file converted from NASA dataset	File Size (bytes)
NISE_SSMIF11_19911227.h5	2M
CER_ES8_Terra-FM2_Test_SCF_016011.20000830.subset_70_20_-140_-40.20001012_204110Z.h5	76M
98034001632_GOES08_IMAGER.h5	24M
avhrr8kmmonthly.h5	24M
balloon_sp.h5	51k

The demonstration implemented simple browsing of a collection of HDF5 files from a standard Web browser. The JSP accessed the HDF5 data, called programs, and generated HTML which was sent to the client. A typical sequence of operations was:

1. Select “hdf5 file” and send the request
2. Get the hdf5 file list, select file “tattr.h5”, request converting to XML file
3. Get the converting result, request the list of XML file
4. Get the XML file list, select “tattr.h5.xml”, request transforming to html file
5. Get the transformed result, request the transformed HTML file.
6. Get tattr.h5.xml.html file

The JSP functions are written special web pages and JavaBeans. The bean is the middleware between data library and JSP. In principle there can be several layers of beans and Java classes to accomplish complex task, and the Java beans can access CORBA services.

We developed a sequence JSP pages that converts files from one format to another with a single click of a Web browser. We demonstrated the following conversions:

- HDF4 to HDF5 (used the *h4toh5* utility).
- HDF5 to XML (used the `ncsa.hdf.io.XMLWriter` class)
- XML to HTML (used Xalan and an XSL style sheet)
- XML to HDF5 (used the `ncsa.hdf.tools.h5gen.H5Gen` class)

2.4. Results and Discussion

The results of this experiment show that we could receive request from a standard Web browser, invoke corresponding JavaBeans classes used in JSP, access the HDF5 library using JNI. We also showed that the JavaBeans could invoke a binary utility program, *h4toh5*.

These technologies could be used to design and implement an application of remote scientific data access with functions similar to CGI-based servers, such as DIAL [7] and SDB [1]. JSP and JavaBeans are flexible and easier to use than CGI.

There are many ways to use JSP and XML together. We could generate XML with JSP, as in this experiment, generate beans from XML, and transform XML into JSP [4, 5]. These ideas need to be investigated further.

Also there are other issues essential to our design. Since scientific data files usually are large and user may not want to acquire entire data at one time, we have to make the application efficient. Using XML technology, Xlink, Xpointer, and XPath could allow partial data access [8, 9]. Also we may need to use some sort of cache techniques to store XML objects in memory.

3. Experiment 2: HDF5 Java/CORBA Experiments

3.1. Introduction

This project investigates using CORBA technology with Java and HDF5 file. The overall approach is to have data access done by a CORBA servant written in C/C++, while browsing and presenting the information in Java (or any other language). This eliminates the need for the JNI, and so avoids the problems discussed in the Introduction.

In this approach, there would be a CORBA server which exports a view of HDF5 files as objects, and accesses the files using the HDF5 library. The CORBA server would be written in C++ and would link to the HDF5 library. It will implement whatever thread-safety and other details need to be managed, using the regular HDF5 library, C/C++, and CORBA.

Clients can access these objects from the server locally (from another process) or remotely across the network. The client can be written in any language. Furthermore, the Java Virtual Machine contains an ORB, so Java classes can directly communicate with CORBA objects as if they were Java. For instance, we could re-implement the H5View as a pure Java application, using CORBA rather JNI. Of particular interest would be light-weight Java applets that could run in a Web browser. This also gives us a fast way to support other languages that can readily use CORBA, including Lisp, Smalltalk, and Python.

3.2. Environment setup

There are several proprietary and free implementations of CORBA [24]. We used the free CORBA from ORBacus [10] which is widely considered the best. Note that the CORBA server requires CORBA for development and to run. The requirements for the client depend on the programming language.

We have tried the earlier version of ORBacus (3.3.4) and found that it is not as well supported as the updated version. Finally we chose ORBacus 4.1.0 since it is fully compliant with CORBA 2.3 standard, and supports POA (Portable Object Adapter) and OBV (Object by Value).

We set up OB-4.1.0 (supports C++) and JOB-4.1.0 (supports Java) on both Linux (Red Hat 7.1) and Solaris 2.7 platforms. ORBacus 4.1.0 comes with test and demo packages for testing the environment setting and showing demos of a series of small, distributed applications.

Note that it requires gcc2.95.3 (or above) and jdk1.3 (or above) to configure and install ORBacus 4.1.0.

3.3. Software Development

The communication between a CORBA server and its clients is based on a common interface, which is essentially a contract between the server and its potential clients. To develop and build a CORBA application, there are three phases required:

- Design of the CORBA interfaces, coded in CORBA IDL.
- Design of the implementation modules to implement the CORBA interfaces.
- Design of client programs to remotely invoke the server.

Design of the IDL interface is the first step of the development of a CORBA application. The IDL interface describes CORBA objects. The IDL for a server specifies the modularized object interfaces, attributes and operations available for that interface. IDL interface are programming language neutral.

The IDL file needs to be compiled with an IDL compiler and converted to its associated representation in the desired programming language according to the standard language binding. This step is required on both of the server and the client sides, prior to the implementation.

To implement the server, first we need to implement the interfaces or objects in the IDL using the desired programming language. Next is to set up the server. The basic steps for setting up a server are:

- Initialize the ORB.
- Create and setup the POA, activate the POA manager.
- Activate objects
- Wait for client requests.

The client implementation is quite simple and easy. The CORBA object implementations are completely transparent to the clients, client just invoke the operations of a CORBA object like its methods of a local object. Also before sending a request to the server, clients need to initialize the ORB in order to communicate with the server.

In this experiment, we designed the IDL according to features we want to demonstrate (see next page), compiled it into C++ for the server and Java for the clients.

There are two server implementation issues beyond this experiment that we would like to address here.

First is the versioning problem. As we mentioned, the IDL file is the communication contract between server and clients. The interface normally is not subject to changing once the development of the system or application is finished. Once it changed, both of the server and client implementation need to be modified or even redesigned. However, new features or changes might be needed for the new version of a system. In order to support the old or previous versions of the IDL interface in the new design, the versioning problem is raised. Applying extension interface pattern is one approach to solve the versioning problem.

The second issue is the structural mechanisms for associating an implementation class with the skeleton class. One is using inheritance approach, while the other is tie approach. The first approach is simple and easy to implement, where the implementation directly extends the skeleton class. The second approach uses delegation to forward requests to the implementation object, which is also referred as an instance adapter or delegation. The tie approach allows the implementation class to inherit from an application-specific class and one class providing the implementation of more than one IDL interface. For Java implemented server, tie approach make the implementation of inherit IDL interfaces much easier.

Even we did not have the versioning problem or use the tie approach in this experiment, the above issues will be useful for the future projects.

3.4. Experiments with HDF

There are two phases of this CORBA/Java and HDF experiment.

The first phase was just implementing a simple CORBA client and server that the client could successfully invoke the operation to access HDF library, such as open a HDF5 file on the server side.

Based on the first phase experiment and the necessary understanding of how CORBA works, we designed and implemented a demonstration application on the second phase.

According to the HDF file structure, we created CORBA objects corresponding to each HDF C++ object, including H5File, H5Group, H5Dataset, and H5Datatype. The following is the IDL interfaces:

```

module HDF5{
    struct ObjInfo{
        long oid;
        long fid;
        long type;
        string name;
        long numOfMembers;
    };

    typedef sequence<long> Sizes;

    struct DatasetInfo{
        string name;
        long numOfDim;
        Sizes dimSizes;
        Sizes dimMaxSizes;
        long dataTypeClass;
        long dataTypeSize;
        long long storageSize;
    };

    struct DatatypeInfo{
        string name;
        long type;
        long size;
    };

    interface H5Obj{
        void getObjInfo(out ObjInfo obji);
        void getNumAttributes(out long attrNum);
        void openAttributes (out long attr);
    };

    interface H5DatasetObj:H5Obj{
        void getDatasetInfo(out DatasetInfo dsinfo);
        long long getStorageSize();
    };

    interface H5GroupObj:H5Obj{
        ObjInfo getGMember(in long fid, in string gName, in long index);
        void openGroup(in string gName, out H5GroupObj h5gobj);
        void getDatasetInfo(in string dsName, out DatasetInfo dsinfo);
        void openDataset(in string dsName, out H5DatasetObj h5ds);
        void getDatatypeInfo(in string ttName, out DatatypeInfo dtinfo);
    };

    interface H5FileObj{
        void getTOC(out ObjInfo obji);
        void openRootGroup(out H5GroupObj h5gobj);
    };

    interface H5FileAccess{

```

```

void openH5File(in string fileName, out H5FileObj h5fobj);
void numOfGMembers(in long fileId, in string gName, out long gNum);
string closeH5File(in long fileId);
};

};

```

We implemented the CORBA server in C++ on Linux platform and created a Java application client on Solaris environment. The GUI based application demonstrates the basic features of the server, such as opening and describing HDF5 files, navigating the Group hierarchy, and showing the metadata of dataset, and data type.

We used Portable Object Adapter to implement the objects. The POA is the intermediary between the implementation of an object and the ORB. It dispatches the request to corresponding servant. Each POA has a set of policies that define its characteristics. Using POA, at least one POA object must exist on the server. In this experiment, we only used the root POA and pass its reference to each object. Figure 3 shows the role of POA between the ORB and the server application.

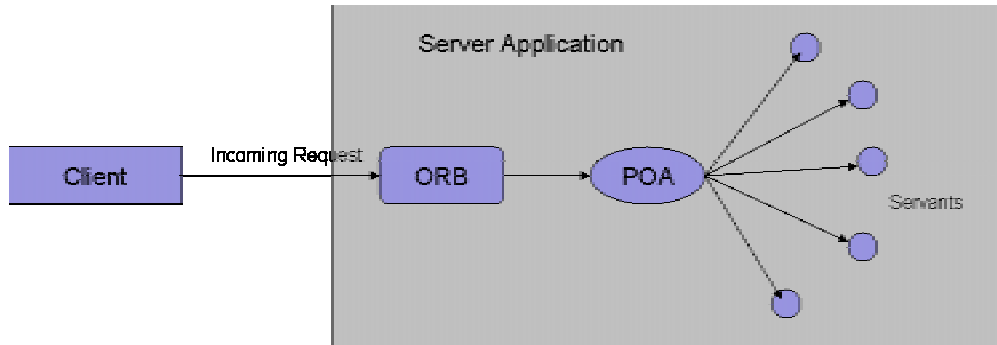


Figure 3. POA – the Mediator between ORB and Server Application

Each HDF5 object is represented by a CORBA object on the server side. References to the objects are passed to the client, which invokes methods remotely to access the objects.

Further we implemented a Java applet on the client side that is able to browse the HDF5 file by navigating the CORBA objects. We used signed applet to be able to invoke those methods from a web browser. The requests were sent from the browser to the web server where the CORBA client resides and further to the CORBA server.

3.5. Summary and Discussion

In this experiment, we developed and deployed the CORBA server in C++ and created a light weighted client in Java. We installed ORBacus 4.1.0 for the server and used JDK built in CORBA package for the client. The structural components of this experimental project are shown in Figure 4.

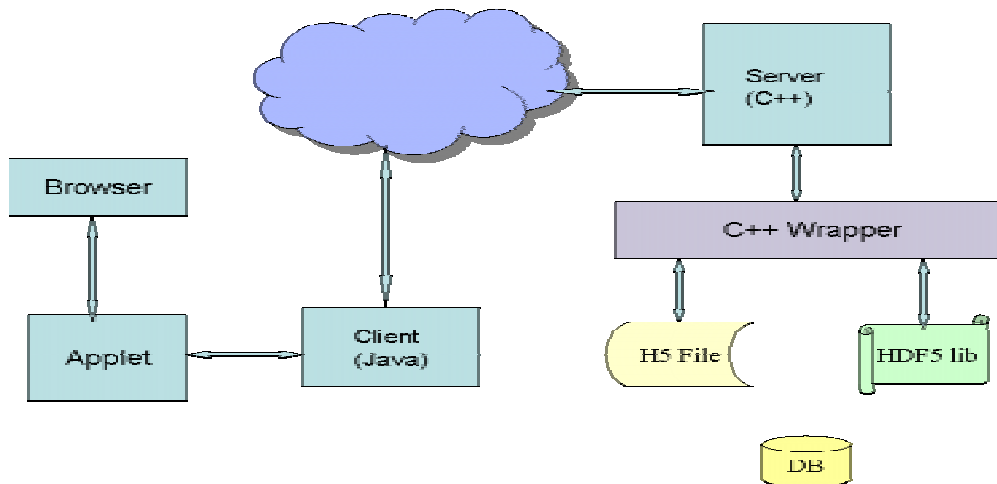


Figure 4. The Components of CORBA-HDF5 project

As an experimental project, only limited functionalities are explored and implemented. Those functionalities are listed in Table 3.

During the experimental period, the server was solid and stable. It was never crashed. The server handles multiple simultaneous requests properly. Meanwhile, the communication between C++ wrapper objects and HDF5 library was partially tested and functions well.

Due to the limitation of the functionality in this project, there many issues and questions remain to be explored. The first issue is how to send data of large dataset to client, by sending the entire data or just reference? Since the data usually are huge and multi dimensions, the representation of the data is a crucial issue, especially when sending back to client via ORB and networking. The second issue is how to support the write access. Current project only involves read access and is relatively easy to design and implement. To support write access, thread safe and synchronization will be the major part to be considered. The third issue is the security issue. Authorization and authentication mechanism will be needed and apply it when clients connect to server and invoke requests. The next issue would be exception handling. For the experimental project, we used the CORBA internal exceptions. For future project, we need to create the necessary exception handling system.

Since this project only demonstrates very limited functionality and mostly provides the metadata of HDF5 file, such as the Table of Content (TOC) of file, group, and dataset, the efficiency and performance issue of using CORBA remains uncertain. However, theoretically the networking speed will be the major factor to affect the performance.

In summary, this experimental project opened one path for remotely accessing and browsing HDF5 file in a distributed system with languages transparency.

Table 3. The CORBA-HDF5 project demonstrated functionalities

	Function	Client Sends	Client Receives
1.	open an H5 File	file name	H5FileObj (reference of CORBA object)
2.	get file Table of Contents	reference of H5FileObj	ObjInfo (CORBA object)
3.	open Root Group	group name ("/" root)	H5GroupObj (reference of CORBA object)
4.	get group TOC	reference of H5GroupObj	ObjInfo (CORBA object)
5.	iterate through the members of a Group	group name	H5Obj (reference of CORBA object)
6.	open a dataset	path, dataset name	H5DatasetObj (reference of CORBA object)
7.	get dataset info	path, dataset name	ObjInfo (CORBA object)
8.	get datatype info	path, datatype name	ObjInfo (CORBA object)
9.	get dataset storage size	dataset reference	storage size (CORBA Long)

4. Discussion

Both the JSP and the CORBA projects investigated the different ways for remotely browsing HDF files. Each way has its own limitations. By using JSP only, we have to use the Java wrapper functions to access the HDF library via JNI. As we mentioned in Section 1, Introduction, JNI has certain limitations and flaws that are what we want to avoid. If we use the approach of the second project, the elimination of using JNI is accomplished by using C++ wrapper functions with the server application. However, the presentation of data in applet is not flexible and can not be customized by clients.

Now a question is raised. Could we combine the two approaches together to eliminate the major limitations of each approach? The answer is certainly we can. In Figure 1, it shows that the JSP communicates to Java beans and the beans could associate with other Java applications. We could design JSP pages and the supporting beans that communicate with a Java client of the CORBA server that is implemented in C++. The requests from the JSP page will be sent to where the JSP engine resides, forwarded to the CORBA client via Java beans. The CORBA client will invoke the remote methods on the CORBA server side and get the desired data back and send the data back to JSP page, which will be displayed in the web browser.

By using JSP, we could browse and display HDF files dynamically with unlimited flexibility. Using CORBA server implemented in C++ as the source of data provider, we could make a solid and powerful server. Combining the two approaches, we could build up a powerful and scalable system with easy customization for users.

Both JSP and CORBA have strong portability with using XML technology. For the future project, how to combine XML with JSP, CORBA together would be the most beneficial area to explore and investigate. We may also need to consider using XML to represent the data of HDF and passing XML string back to JSP.

Another suggestion to the future projects is to adapter visualization techniques to display HDF file and data. Using visualization with XML, JSP, and CORBA technologies will be the most interesting and challenge project for us.

5. Acknowledgements

Kun Yan conducted these experiments as a Research Assistant at the N.C.S.A.

This report is based upon work supported in part by a Cooperative Agreement with NASA under NASA grant NAG 5-2040. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Aeronautics and Space Administration.

Other support provided by NCSA and other sponsors and agencies [6].

6. References

1. "Scientific Data Browser (SDB)", <http://hdf.ncsa.uiuc.edu/sdb/sdb.html>
2. JavaServer PagesTM (JSPTM), <http://java.sun.com/products/jsp/>
3. Alex Chaffee, "Using XML and JSP together", http://www.javaworld.com/javaworld/jw-03-2000/jw-0331-ssj-jspxml_p.html
4. Kammie Kayl, "Creating web services with Java technology and XML", <http://www.sun.com/software/cover/2001-0530/>
5. Timothy Eden and Ed Ludke, "Introducing JavaServer Pages", <http://www.devx.com/upload/free/features/javapro/2000/04apr00/te0004/te0004-1.asp>
6. <http://hdf.ncsa.uiuc.edu/acknowledge.html>
7. "DIAL (Data and Information Access Link)", <http://dial.gsfc.nasa.gov/>
8. "XML Pointers, XML Base and XML Linking", <http://www.w3.org/XML/Linking>
9. "XML Path Language", <http://www.w3.org/TR/xpath>
10. "ORBacus For C++ and Java (Version 4.1.0)", <http://www.orbacus.com>
11. "Object Management Group, <http://www.omg.org>
12. Michi Henning and Steve Vinoske, *Advanced CORBA Programming with C++*, Addison-Wesley, 1999.
13. "Introduction to CORBA", <http://java.sun.com>
14. "NCSA HDF", <http://hdf.ncsa.uiuc.edu>
15. Robert E. McGrath, "A Scientific Data Server: The Conceptual Design", 1997, http://hdf.ncsa.uiuc.edu/horizon/DataServer/sds_design.html.
16. "NCSA Java HDF Server (JHS)", <http://hdf.ncsa.uiuc.edu/java-hdf-html/jhs/>
17. Robert E. McGrath, "Experiment with XSL: translating scientific data" February, 2001 <http://hdf.ncsa.uiuc.edu/HDF5/XML/nctoh5/writeup.htm>
18. Robert E. McGrath, "An Experimental Comparison of HDF4, HDF5, and XML Representations of the Same Dataset" (May, 2001) <http://hdf.ncsa.uiuc.edu/HDF5/XML/EOSData/h4-h5-xml.htm>
19. "Experiment on JSP, XML, and HDF" <http://hdf.ncsa.uiuc.edu/HDF5/XML/JSPEperiments/index.html>
20. "HDF (4.x) and HDF5", <http://hdf.ncsa.uiuc.edu/h4toh5/>
21. "Apache httpd", <http://www.apache.org/dist/httpd/>
22. "The Jakarta project", <http://jakarta.apache.org/>
23. HDF5.dtd, <http://hdf.ncsa.uiuc.edu/DTDs/HDF5-File-1.4.txt>
24. Free CORBA page, <http://adams.patriot.net/~tvalesky/freecorba.html>