Writing an OPeNDAP Server
Document version 1.1

James Gallagher

July 20, 2002

Contents

1 Preface

Writing your own OPeNDAP server is one way to serve data to clients that understand the Data Access Protocol (DAP). This tutorial describes the writing of such a server.

2 Before you write any code

Even if the data you want to serve are stored in some idiosyncratic format, or accessed using an API that you or your group have developed, there may already be a server which, with the correct configuration, can work with your data.

The DODS project has developed two servers which can be customized to read different types of data. One, the FreeForm server can be used to provide table and array data, with some restrictions on the data's storage format. The JGOFS server can only serve table-like data but has a more flexible — and more complicated — customization scheme. More introductory information about these two servers can be found in the Server Installation Guide.

2.1 The FreeForm server

The FreeForm server is configured for a specific dataset using a 'format specification file .' Detailed information about writing these files for your data may be found in the FreeForm server guide. The main limitation of the FreeForm server is that it requires that all information be rigidly column-oriented. For example, consider the two ASCII data files shown below:

```
  1 one 13.4                           1 one   13.4
  2 two 27.8                           2 two   27.8
  3 three 17.4                         3 three 17.4

ASCII data that cannot                 Data that can
be served with FreeForm
```

The first file cannot be served by the FreeForm server because the third value in each record does not start at the same column position. The same data can be served only if each datum in all the records 'lines up.' In addition to ASCII data, the FreeForm server can also serve binary data and DBase files. The FreeForm server cannot serve CSV files.

2.2 The JGOFS server

The JGOFS server can serve data that is logically structured in a table-like fashion. By table-like, we mean that the data are organized in row and column form with the additional twist that the rightmost column can itself be a table; it's not limited to the standard simple data types such as Integer or floating point numbers. This enables the JGOFS server to efficiently serve data that are hierarchical without duplicating values at the outer levels. Compare JGOFS' nested tables with a regular flat table scheme:

```
station depth temp                  station depth temp
1       10    10.2                  1       10    10.2
        20    10.3                  1       20    10.3
        30    10.0                  1       30    10.0
2       10    10.2                  2       10    10.2
        20    10.35                 2       20    10.35
        30    10.0                  2       30    10.0

   Nested Tables                 A Flat Table Duplicates Values
```

The JGOFS server can also read data that's organized in many files, making access to the individual files seamless. In practice this is quite powerful. If information about stations, continuing with the example above, is stored in one file and information about each depth/temperature sounding is stored in separate files, JGOFS can easily be configured to serve these data. In fact, the JGOFS server comes with a standard 'method' that can serve data stored in just this configuration as long the data are stored as ASCII values.

However, JOGFS is far more flexible than just a scheme to serve ASCII data. The JGOFS server is customized by writing a specialized access method for a new data storage format. For more information about the JGOFS system see the JGOFS Data System Overview. There is also a detailed method writing guide with examples available for JGOFS.

3 Writing your own OPeNDAP server

If neither of the available flexible data-readers can work with your data, or if you wish to add some features that are not otherwise available, you may find it advisable to write your own OPeNDAP server.

3.1 Choose a language

It is possible to take the DAP specification and implement a server which DAP-aware clients can use as a data source. For example, the data server hosted by the IRI/LDEO Climate Data Library at Columbia University is such a server. However, server writers don't have to work from the DAP specification. The DODS Project has developed two separate object-oriented class libraries which, along with other software we provide, can greatly simplify building a server in most cases.

The DODS project provides both a C++ and a Java implementation of the DAP. Each library includes both the classes that implement the various objects which comprise the DAP and support software that handles the mechanics of processing inbound requests and generating the correct responses.

To choose one of the toolkits, several factors should be weighed. First, with which of the two programming languages are you most comfortable? Also to be considered are: What type of computer will the server run on. Java is equally supported on win32 and Unix (and mac, in all probability) while the C++ code for server development is supported only on Unix. (The client libraries are supported on Windows.) If you have an API that can read the data, will it be easier to use it from C++ or Java? Lastly, the Java servers are implemented as servlets and typically spend less time on startup tasks than the C++ servers which use the CGI mechanism. If you anticipate many small requests, then you can expect noticeable performance improvements with the Java code, while larger average requests will mitigate this difference.

> **NOTE:** Developers at UCAR have implemented DODS servers as Apache httpd modules; these DODS servers effectively run as Unix daemons and thus have none of the startup performance issues of CGI programs. These servers are noticeably faster—about one order of magnitude—for very small requests. In the future we plan to incorporate this software in our general distribution, contact technical support or the dods-tech list for information/help.

3.2 Server architecture

The essence of the DODS server architecture is that a collection of programs are used to handle various requests made to the servers. In addition to these 'handler' and 'service' programs there's also a dispatcher that interfaces to an http daemon. The actual requests are made to the web daemon which then passes them along to the dispatcher. The dispatcher examines the request and decides which handler or service program should process it and how that program should be passed parameters extracted from the request.

> **NOTE:** The C++ software works exactly as described above; the Java code is the same in principle but slightly different in practice, since it's based on servlets.

While the C++ toolkit uses an architecture based on CGI and the Java toolkit uses servlets, both share many characteristics. If you understand how the servers are built, it will be easy to see how your own server can be implemented with minimal effort. The Server Installation Guide's section on Server Architecture provides an excellent description of the CGI-based (C++) servers. The Server Installation Guide contains a short how-to that covers setting up the Java software. It also explains the software needed to run the servlet-based DODS servers.

4 The DAP Architecture

The DAP can be thought of as a layered protocol composed of MIME, HTTP, basic objects, and complex, presentation-style, responses.

4.1 The DAP uses HTTP which in turn uses MIME

Clients use HTTP when they make requests of DAP servers. HTTP is a fairly straightforward protocol (General information on HTTP, The HTTP/1.1 specification in HTML). It uses MIME documents to encapsulate both the request sent from client to server and the response sent back. This is important for the DAP because the DAP uses headers in both the request and response documents to transfer information. However, for a programmer who intends to write a DAP server, exactly what gets written into those headers and how it gets written is not important. Both the C++ and Java class libraries will handle these tasks for you (look at the DODSFilter class to see how). It's important to know about, however, because if you decide not to use the libraries, or the parts that automate generating the correct MIME documents, then your server will have to generate the correct headers itself.

4.2 The DAP defines three objects

To transfer information from servers to clients, the DAP uses three objects. Whenever a client asks a server for information, it does so by requesting one of these three objects (note: this is not strictly true, but the whole truth will be told in just a bit. For now, assume it's true). These are the Dataset Descriptor Structure (DDS), Dataset Attribute Structure (DAS), and Data object (DataDDS). These are described in considerable detail in other documentation. The Programmer's Guide contains a description of the DDS and DAS objects. These objects contain the name and types of the variables in a dataset, along with any attributes (name-value pairs) bound to the variables. The DataDDS contains data values. We have implemented the SDKs so that the DataDDS is a subclass of the DDS object that adds the capacity to store values with each variable.

COADS Climatology	DAS	DDS
NASA Scatterometer Data	DAS	DDS
Catalog of AVHRR Files	DAS	DDS
AHVRR Image	DAS	DDS

The DAP models all datasets as collections of variables. The DDS and DataDDS objects are containers for those variables. How you represent your dataset using the three objects and the variable's data type hierarchy is covered in section 7.

4.3 The DAP also defines services

In the previous section we said that the DAP defined three objects and all interaction with the server involved those three objects. In fact, the DAP also defines other responses. They are:

ASCII Data can be requested in CSV form.

HTML Each server can return an HTML form that facilitates building URLs.

INFO Each server can combine the DDS and DAS and present that as HTML.

In each case the server's response to these requests is built using one or more of the basic three objects. Here are some links to various datasets' ASCII, HTML and INFO responses:

COADS Climatology	ASCII for the SST variable	HTML	INFO
NASA Scatterometer Data	ASCII for wind speed and direction	HTML	INFO
Catalog of AVHRR Files	ASCII for values within a date range	HTML	INFO
AHVRR Image	ASCII for the SST	HTML	INFO

4.4 Parts of the server you don't have to write

You do not have to write handlers for the ASCII, HTML or INFO responses because the DODS server includes software that generates these using the DAS, DDS and DataDDS objects. In addition, if you follow a simple rule about how you name the programs that generate the object responses, you'll be able to fit these within the existing dispatch software and can avoid writing that as well. The rule is that the three objects are generated by programs named *name*`_das`, *name*`_dds` and *name*`_dods` (the last one generates the DataDDS object). The *name* can be any text. In practice, it should be short and describe the data with which it's designed to work.

Below is a snapshot of the directory which holds the programs that make up the DODS servers on my development computer. The ASCII response is generated by the `asciival` program, The HTML and INFO responses are generated by the `www_int` and `usage` programs. You can also see the dispatch program (`nph-dods`) as well as the DAS, DDS and DataDDS handlers for the netCDF (nc), HDF (hdf), Matlab (mat), JGOFS (jg) and FreeForm (ff) servers.

```
[jimg@zanzibar etc]$ ls
aclocal.m4           ftp_dods_source.html   MIME/
asciival*            handler_name.pm        nc_das*
ChangeLog            hdf_das@               nc_dds*
check_perl.sh*       hdf_dds@               nc_dods*
common_tests.exp     hdf_dods*              nightly_dods_build.conf
config.guess*        HTML/
nightly_dods_build.conf.example
config.sub*          HTTP/                  nightly_dods_build.sh*
COPYRIGHT            INSTALL-clients        nph-dods*
CVS/                 INSTALL-matlab-client  nph-dods.in*
cvsdate*             installServers         printenv*
def*                 INSTALL-servers        README
deflate*             install-sh*            README-Matlab-GUI
depend.sh*           jg_das*                tar-builder.pl*
DODS_Cache.pm        jg_dds*                test-dispatch.sh*
DODS_Dispatch.pm     jg_dods*               ud_aclocal.m4
dods.ini             jgofs_objects_readme*  update-manifest.pl*
ff_das*              localize.sh*           update-manifest.pl~*
ff_dds*              LWP/                   usage*
ff_dods*             Makefile.common        usage~*
FilterDirHTML.pm     mat_das*               usage-jg*
ftp_dods_binary.html mat_dds*               www_int*
ftp_dods_ml_gui.html mat_dods*
[jimg@zanzibar etc]$
```

5 Getting ready to write your components

The three object handlers are normally implemented in three separate programs. Each program has a `main()` function that looks like:

```
#include <iostream>
#include <string>

#include "DDS.h"
#include "cgi_util.h"
#include "DODSFilter.h"

extern void read_descriptors(DDS &dds, const string &filename);
throw  (Error);

int
main(int argc, char *argv[])
{
    DDS dds;
    DODSFilter df(argc, argv);

    try {
        if (!df.OK()) {

df.print_usage();
            return 1;
        }

        if (df.version()) {

df.send_version_info();
            return 0;
        }

// Read the netCDF file dataset descriptor in memory
        read_descriptors(dds,  df.get_dataset_name());
        df.read_ancillary_dds(dds);
        df.send_dds(dds, true);
    }

    catch (Error &e) {

        set_mime_text(cout, dods_error,
        df.get_cgi_version());

        e.print(cout);
        return 1;
    }

    return 0;
}
```

Most of the software is boilerplate. The first two lines of `main()` are shown here:

```
DS dds;
DODSFilter df(argc, argv);
```

These declare an instance of DDS, to be used a little later as well as an instance of DODSFilter. The latter is used to parse command line arguments fed to the program by the dispatch script. By using this class and the dispatch script, you can assume that the correct options and arguments will be passed into your program and parsed. The instance of DODSFilter, `df`, contains accessors for all the switches that the dispatch script might use, so by passing `argc` and `argv` to it you're sure to parse them all.

```
try {
   if (!df.OK()) {
         df.print_usage();
         return 1
   }

   if (df.version()) {
         df.send_version_info();
         return 0;
   }
```

This code calls the DODSFilter invariant to check that the handler was invoked correctly. If a malformed request was made to the server, this will be flagged here and the server will return an error message describing how to submit a correctly formed URL. This also tests to see if the request is for version information. If so, the DODSFilter object prints the server's version number and the handler exits. Just about every server built with our code includes these lines verbatim.

```
// Read the netCDF file dataset descriptor in memory

   read_descriptors(dds, df.get_dataset_name());
   df.read_ancillary_dds(dds);

   df.send_dds(dds, true);
```

These lines are the heart of the handler. Exactly what's going on here will be covered in more detail later. However, each of the three object handlers contains similar code that builds the object to returned as the response and then passes that object to the `DODSFilter::send_dds`, `send_das` or `send_data` method, depending on the type of object to be returned.

```
catch (Error &e) {
        set_mime_text(cout, dods_error, df.get_cgi_version());
        e.print(cout);

        return 1;
       }

       return 0
```

Rounding out the program is a catch block that picks up exceptions thrown by the any of DAP library code. The DAP library throws two types of exceptions, `Error` and `InternalErr`. The latter is a subclass of `Error`, so catching just Error will get everything. Note that you should also catch `bad_alloc` exceptions at this level (the library does not) unless you catch them inside the function or method that builds the DDS, DAS or DataDDS. If `e` is an `InternalErr`, then when it prints, you'll see information about the file and line number where the problem was detected. Regular `Error` objects print something that's more useful to users. By calling the `set_mime_text` function (see the file cgi_util.cc) you're sure that the error message will be returned to the client in a form that both web browsers and more sophisticated clients can use.

6 Subclassing the data types

The DAP defines a data type hierarchy as the core of its data model. This collection of data types includes scalar, vector and constructor types. Most of the types are available in all modern programming languages with the exceptions being Url, Sequence and Grid. In the DAP library, the class `BaseType` is the root of the data type tree.

6.1 A quick review of the data types supported by the DAP

The DAP supports the common scalar data types such as Byte, 16- and 32-bit signed and unsigned integers, and 32- and 64-bit floating point numbers. The DAP also supports Strings and Urls as basic scalar types. The DAP includes two vector data types, Arrays (of unlimited size and dimensionality) and Lists. Lists in the DAP must be type-homogeneous; it does not support Lists of Lists. The DAP also supports three type-constructors: Structure, Sequence and Grid. A Structure on the DAP mimics a struct in C. A Sequence is a table-like data structure inherited from the JGOFS data system. It can be used to hold information that might be stored in relational databases or tables, either flat or hierarchical. The JGOFS, FreeForm and HDF servers all use the Sequence data type. Lastly, the Grid data type is used to bind an array to a group of 'map vectors,' single dimension arrays that provide non-integral values for the indices of the array. The most typical use of a Grid is to provide latitude and longitude registration for some georeferenced array data (e.g., a projected satellite image). The DAP does not have a pointer data type, but in some cases the Url data type can be used as a pointer to variables between files. More information about the DAP's data type hierarchy is given in the Programmer's Guide.

6.2 Creating the subclasses

When you start building a DAP server, the first thing you must do is create a collection of data type subclasses. That is, each of the leaf classes in the preceding class diagram must be subclassed by your server. This is pretty easy since a good bit of the work is rote.

First we'll illustrate the parts that are mechanical. Here's an example from the Matlab server. The class is the Byte class. In the case of the matlab server, this class doesn't do anything beyond the bare minimum, so it's a good starting point:

```
Byte *
NewByte(const string &n)
{
    return new MATByte(n);
}

MATByte::MATByte(const string &n) : Byte(n)
{
}

BaseType *
MATByte::ptr_duplicate()
{
    return new MATByte(*this);
}

bool
MATByte::read(const string &)
{
    throw InternalErr(__FILE__, __LINE__, "Unimplemented read method
    called.");
}
```

To create a child of any of the data type leaf classes, you must define three methods and one function. Let's talk about the function first. The function `NewByte` is what Meyers calls a 'virtual constructor.' It's similar to a low-budget factory class ("low-budget" because it's not a class). This function is undefined in the DAP library but is used there when creating instances of Byte. Thus the function `NewByte` is an interface that can be used to create instances of Byte without knowing in advance the static type of the object that will be created. If all this sounds a little weird, just remember that your Byte, Int16, ... Grid classes — whatever they may be called — must all contain an implementation of this function and they should all return a pointer to an instance of your child classes. In this case, it's an instance of the MATByte class. If you look in the files for the Matlab server, you'll see that Grid returns a pointer to a new MATGrid, and so on.

Second, a constructor must be implemented and should take the name of the variable as its sole argument.

Third, your child classes should also define the ptr_duplicate() method. This method returns a pointer to a new instance of an object in the same class. Occasionally, in the DAP library, objects are declared with pointers specified as `BaseType *`. If the `new` operator was used to copy such an object, the copied object would be an instance of BaseType (the static type of the object) not the type of the thing referenced (the dynamic type)[1]. By using the `ptr_duplicate()` method the DAP library is sure that when it copies an object, it's getting an instance of the subclass defined by your server.

Finally, each of the child classes must provide an implementation of the `read` method. This method is called by code in the DAP library to read values from the data set. It will be explained in more detail when we get to building DataDDS responses. For now, it's enough to know that if a particular server has no use for a given data type (it happens that the Matlab server will never need to create an instance of Byte, because Matlab 5 files can only store float64 matrices) this method should throw an InternalErr object.

7 Implementing the DDS object

Building a DDS object is the heart of writing your own OPeNDAP server. This object will be used to generate the DDS response and it will be the basis of the DataDDS response. You have to do two things to accomplish building the DDS. First you must decide how the variables that comprise your dataset can be represented using the data type hierarchy that is part of the DAP. Once you have done this, you need to write code that can build an instance of DDS for your dataset. In practice the hardest part of this the first part; once you know how to map variables in your dataset to the DAP data types, writing code to build the DDS instance is easy.

Many data sets are actually a representative of large group. In some cases there may be an API that can read the datasets and there may even be a formal data model. In such a case you're best off using the API and writing general code to build the DDS while performing a depth-first scan of the variables in the dataset.

Here's how the Matlab server builds a DDS object:

[1] This is the oft discussed phenomenon of 'slicing,' see Meyers, Stroustrup, et cetera for a complete explaination.

```
void
read_descriptors(DDS &dds_table, string filename)
{
    MATFile *fp;
    Matrix *mp;

    // dataset name
    dds_table.set_dataset_name(name_path(filename));

    fp = matOpen(filename.c_str(), "r");
    if (fp == NULL)
        throw Error(string("Could not open the file: ") + filename);

    // Read all the matrices in file
    while ((mp = matGetNextMatrix(fp)) != NULL) {

      // String types are used as attributes
        if(mxIsNumeric(mp)) {
            if(mxIsComplex(mp)) {
                string Real = (string)mxGetName(mp) + "_Real";
                // real part
                MakeMatrix(&dds_table, Real, mxGetM(mp),mxGetN(mp));

                string Imag = (string)mxGetName(mp) + "_Imaginary";
                // imaginary part
                MakeMatrix(&dds_table, Imag, mxGetM(mp),mxGetN(mp));
         } else
                MakeMatrix(&dds_table, (string)mxGetName(mp), mxGetM(mp),
                            mxGetN(mp));
         }

          mxFreeMatrix(mp);
    }
    matClose(fp);

    return true;
}
```

This function iterates over all the variables in the Matlab file named by `filename` and creates a variable in the DDS for each numerical array in the dataset. Matlab does not have the notion of attributes bound to specific variables, but it is often the case that attribute information is present in string variables, something for which this code checks. However, this function simply ignores the string variables since attribute information is the job of a different object. Of course, a function could be written to build both objects at the same time…

```
void
MakeMatrix(DDS *dds_table, string name, int row, int column)
{
    Array *ar;
    string DimName;
    size_t pos;

    // complex matrices have common rows and columns
    if ((pos = name.find("_Real")) != name.npos)
        DimName = name.substr(0, pos);
    else{
        if ((pos = name.find("_Imaginary")) != name.npos)
            DimName = name.substr(0, pos);
        else
            DimName = name;
    }

    BaseType *bt =  NewFloat64(name);
    ar = NewArray(name);
    ar->add_var(bt);
    ar->append_dim(row,DimName+"_row");
    ar->append_dim(column,DimName+"_column");

    if (!dds_table)
        throw InternalErr(__FILE__, __LINE__, "NULL DDS object.");

    dds_table->add_var(ar);
}
```

This function has two main parts, the first, which is of less interest to this tutorial, checks to see if the matrix holds complex numbers and does some special stuff if it does. The second part creates a new array of 64 bit floating point numbers. Here's the code:

```
BaseType *bt = NewFloat64(name);
ar = NewArray(name);
ar->add_var(bt);
ar->append_dim(row, DimName+"_row");
ar->append_dim(column, DimName+"_column");
```

The first line creates a new instance of the `Float64` data type and assigns it to a `BaseType`, the parent of all the data types. There's no reason it couldn't be bound to an instance of `Float64`, but in a server where there might be many types of arrays, it is easier to use a pointer to a more general object. The second line creates a new Array instance and the third line binds the `Float64` object to the new `Array` object, making the array an array of `Float64`s. The last two lines set the sizes of the Array's dimensions. Because instances of Array occur frequently, it is a good idea to be familiar with the `Array` and `Vector` classes (`Vector` is the parent of both `Array` and `List`).

Finally, the last line of the function,

```
dds_table->add_var(ar);
```

Adds variable `ar` to the DDS.

Note that our Matlab server supports only the data types that can appear in a Matlab 5 file. This means that the only numeric data type supported is a matrix of 64 bit floating point numbers. Strings, as mentioned earlier, are handled specially. In most other servers, the code to build the variables and load them in a DDS object is more complex since it must handle mapping the dataset's different types to the DAP's.

8 Implementing the DAS object

To implement the DAS handler, start with the same boiler-plate code used to create a `main()` for the DDS handler and replace the call to the function that builds the DDS with one that builds a DAS. An example of such a function from the Matlab server is shown below:

```
void
read_attributes(DAS &das_table, string filename)
{
    AttrTable *attr_table = das_table.add_table("MAT_GLOBAL", new AttrTable);

    MATFile *fp = matOpen(filename.c_str(), "r");
    if (fp == NULL)
        throw Error(string("Could not open the file: ") + filename.c_str());

    // Read all the matrices in file
    Matrix *mp;
    while ((mp = matGetNextMatrix(fp)) != NULL) {
        // String types are used as attributes
        if(mxIsString(mp)) {
            // get size
            int X = mxGetN(mp);
            int Y = mxGetM(mp);

            char *str_rep = new char [X*Y+3];

            // quote the string for parser
            *str_rep = '"';
            mxGetString(mp, str_rep+1, X*Y+1);
            *(str_rep + X*Y + 1) = '"';
            *(str_rep + X*Y + 2 )= '\0';

            if (attr_table->append_attr(mxGetName(mp),
                "String",str_rep) == 0) {
                delete [] str_rep;
                mxFreeMatrix(mp);
                matClose(fp);
                throw Error(string("Couldn't output attribute: ")
                            + mxGetName(mp));
            }

            delete [] str_rep;
        }
        else {
            das_table.add_table(mxGetName(mp), new AttrTable);
        }
        mxFreeMatrix(mp);
    }
    matClose(fp);

    return true;
}
```

In this example the server creates a single attribute container called MAT_GLOBAL and loads all the data set attributes into it.

Most data set types (e.g., hdf) have both global attributes (those that apply to the entire data set) and attributes that only apply to a particular variable. Such data sets have both a global attribute container *and* separate attribute containers for each variable[2]. Matlab has only global attributes.

Here's what is going on inside this function:

First a new attribute table object is created and added to the DAS object. An attribute table (AttrTable) is similar to a structure in that it holds other things which may be attributes (typed name-value pairs) or other attribute tables. The DAS is a container for AttrTable objects. Take a look at the documentation for the AttrTable and DAS classes in the Programmer's Reference Guide.

```
AttrTable *attr_table = das_table.add_table("MAT_GLOBAL", new AttrTable);
```

Following the creation of a table for global attributes, the function handles the routine and API-dependent tasks of opening the data set and setting things up to iterate over its variables.

```
MATFile *fp = matOpen(filename.c_str(), "r");
if (fp == NULL)
    throw Error(string("Could not open the file: ") + filename.c_str());

// Read all the matrices in file
Matrix *mp;
while ((mp = matGetNextMatrix(fp)) != NULL) {
```

Inside the `while`-loop that iterates over the data set's variables, we test for string variables. The Matlab server assumes that all string variables in a data set are actually global attributes for the data set (and not 'data' variables). If a string variable is found, the code uses the variable's name and value as the attribute name and value (code that handles the case where a variable is not a string is explained further down).

Attributes are all string-valued in the DAP. That is, even though the DAP supports the full range of scalar data types for attributes, the *values* are stored as strings. The call to `AttrTable::append_attr` adds the attribute tuple (Name, type and value) to the AttrTable instance.

Note that if `AttrTable::append_attr` fails, it returns zero and the code cleans up and throws an exception. I elided that from this sniplet to focus the example.

[2]By convention, global attribute containers which hold values read from the data set has the suffix _GLOBAL; the first part of the container's name is either read from the dataset or is some appropriate string chosen by the server.

```
        // String types are used as attributes
        if(mxIsString(mp)) {
            // get size
            int X = mxGetN(mp);
            int Y = mxGetM(mp);

            char *str_rep = new char [X*Y+3];

            // quote the string for parser
            *str_rep = '"';
            mxGetString(mp,str_rep+1,X*Y+1);
            *(str_rep + X*Y + 1) = '"';
            *(str_rep + X*Y + 2 )= '\0';

            if (attr_table->append_attr(mxGetName(mp), "String", str_rep) == 0) {
\end{vode}

If the variable is not a string, then the Matlab server creates an empty
attribute table for it. This is to conform to the \xlink{DAP 2.0 draft
  specification}{http://www.unidata.ucar.edu/packages/dods/design/dap-rfc-html/dap_32.html}
which states that all variable must have an attribute table, even if it is
empty.

\begin{vode}{sib}
        else {
            das_table.add_table(mxGetName(mp), new AttrTable);
        }
```

9 Implementing the DataDDS object

Building the DataDDS object handler follows the same pattern as before with the DDS and the DAS. In fact, the DDS handler can be modified and used as the DataDDS handler by simply changing the call to `DODSFilter::send_dds()` to a call to `send_data()`. However, before the `send_data()` method will work, we must return to the data type child classes and add more functionality.

In the data type child classes you created we must now implement the method `read()`. This method will be called by the DAP software that sends data values. To understand how the `read()` method will be used, it's instructive to look at the code that calls it. In the DAP data type classes, each of the scalar, vector and constructor types has a method called `serialize()`. Below is shown Byte's version of this method (Look at the code for DDS if you'd like to see how `serialize()` is used and pay particular attention to `DDS::send_data()`).

> Note: You don't have to write your own version of `serialize()`, this is shown here to provide you with some background information about the role of the `read()` method in building the DataDDS response.

```
bool
Byte::serialize(const string &dataset, DDS &dds, XDR *sink, bool ce_eval)
{
    if (!read_p())
        read(dataset);          // read() throws Error and InternalErr

    if (ce_eval && !dds.eval_selection(dataset))
        return true;

    if (!xdr_char(sink, (char *)&_buf))
        throw Error(
"Network I/O Error. Could not send byte data.\n\
This may be due to a bug in DODS, on the server or a\n\
problem with the network connection.");

    return true;
}
```

The `serialize()` method is broken into three parts:

- Read the data for this variable if that has not already been done. We'll see later that when `read()` successfully completes, it sets a boolean that the DAP library tests with the `read_p()` method. If `read_p()` returns `true` then data for this variable has already been read[3].
- Next the constraint expression (CE) is evaluated. The constraint expression was passed to the server as part of the DataDDS request. Parsing the CE and ensuring it's evaluated correctly is handled for you by the DAP library.
- Finally, if the CE evaluates to `true`, then we serialize the binary data that was read into this instance with the `read()` call at the beginning of this method.

> **NOTE:** CE evaluation actually happens in two phases. In the phase, the expression is parsed. During this process, variables that are 'projected' are marked as such and a linked list of 'selection' nodes is built. The `serialize()` method is called only for variables that are part of the projections (that is, that are to be sent back to the client). The second phase of CE evaluation happens inside `serialize()` when the `DDS::ce_eval` method is used to evaluate the clauses. If all the clauses evaluate to true, then the current variable is sent.

Here's the `read()` method for the Matlab server's `Array` class (`MATArray`). This is by far the most complex looking piece of code in this tutorial, but it's really not very complicated once broken down.

[3] In some cases, data for a variable is read while evaluating the constraint expression. So it can be the case that the values for a variable are read before the `serialize()` method calls `read()`.

```
bool
MATArray::read(const string &dataset)
{
    if (read_p())  // Nothing to do
        return false;

    MATFile *fp = matOpen(dataset.c_str(), "r");
    if (fp == NULL)
        throw Error(string("Could not open the file: ") + dataset.c_str());

    Pix p = first_dim();
    int start = dimension_start(p,true);
    int stride = dimension_stride(p, true);
    int stop = dimension_stop(p, true);

    next_dim(p);
    int start_p = dimension_start(p,true);
    int stride_p = dimension_stride(p, true);
    int stop_p = dimension_stop(p, true);


    // get real part of the complex  matrix
    double *DataPtr;
    Matrix *mp;
    size_t pos;
    if ((pos = name().find("_Real")) != name().npos) {
        string Rname = name().substr(0,pos);
        mp = matGetMatrix(fp,Rname.data());
        DataPtr = mxGetPr(mp);
    }
    else{
        // get Img part of the complex matrix
        if ((pos = name().find("_Imaginary")) != name().npos) {
            string Iname = name().substr(0,pos);
            mp = matGetMatrix(fp,Iname.data());
            DataPtr = mxGetPi(mp);
        }
        else{
            mp = matGetMatrix(fp,name().data());
            DataPtr = mxGetPr(mp); // get the matrix structure
        }
    }

    if (DataPtr == NULL)
        throw Error(string("Error reading matrix"));

    if(start+stop+stride == 0){ //default rows
        start = 0;
        stride = 1;
        stop = mxGetM(mp)-1;
    }
    if(start_p+stop_p+stride_p == 0){ //default columns
        start_p = 0;
        stride_p = 1;
        stop_p = mxGetN(mp)-1;
    }

    int Len = (((stop-start)/stride)+1)*(((stop_p-start_p)/stride_p)+1);

    int Tcount = 0;
    dods_float64 *BufFlt64 = new dods_float64 [Len];

    for (int row = start; row <= stop; row +=3Dstride) {
        for(int column = start_p; column <= stop_p; column+=stride_p) {
            *(BufFlt64+Tcount) = (dods_float64)
            *(DataPtr+row+column*mxGetM(mp));
            Tcount++;
        }
    }

    set_read_p(true);
    val2buf((void *)BufFlt64);
    delete [] BufFlt64;

    mxFreeMatrix(mp);
    matClose(fp);
    return false;
}
```

First, on entry into the method, we check to see if the data have already been read. This can happen if the data were previously needed for the evaluation of the CE. Note that in an earlier version of the DAP library, the return value of `read()` was used to signal whether the method needed to be called again to read more data (`false` indicated that all the data had been read). Now calls to `read()` always get all the data, but the return type is still `bool` because older code checks the return value. In software that uses 3.2 or newer, `read()` should always exit by returning `false` unless it encounters an error, in which case it should throw an exception.

```
if (read_p())  // Nothing to do
    return false;
```

If the data has not yet been read, the method then opens the Matlab data set. Each data source is different, but conceptually, this action has to be performed somewhere. In some cases, the data set would be opened once someplace else and the `read()` methods would access some sort of pointer or other access token.

```
MATFile *fp = matOpen(dataset.c_str(), "r");
if (fp == NULL)
    throw Error(string("Could not open the file: ") + dataset.c_str());
```

Read the data for the variable from the data set. Again, this will vary with each type of data set. In the Matlab server, a complex matrix is represented in the DAP by two different matrices, once with the suffix `_Imaginary` and one with the suffix `_Real`. This code looks at the name of the variable and uses that to find the correct variable and read its values. More complex data sets will probably need a more sophisticated lookup scheme.

```
// get real part of the complex  matrix
double *DataPtr;
Matrix *mp;
size_t pos;
if ((pos = name().find("_Real")) != name().npos) {
    string Rname = name().substr(0,pos);
    mp = matGetMatrix(fp, Rname.data());
    DataPtr = mxGetPr(mp);
}
else{
    // get Img part of the complex matrix
    if ((pos = name().find("_Imaginary")) != name().npos) {
        string Iname = name().substr(0,pos);
        mp = matGetMatrix(fp, Iname.data());
        DataPtr = mxGetPi(mp);
    }
    else{
        mp = matGetMatrix(fp,name().data());
        DataPtr = mxGetPr(mp); // get the matrix structure
    }
}

if (DataPtr == NULL)
    throw Error(string("Error reading matrix"));
```

Once the data have been read from the data set we need to check for sub-sampling that may have been specified by the client and passed to the server via the CE. The CE was automatically parsed by the boilerplate code, but we need to explicitly look at the values because data set types are fairly idiosyncratic about how they use this information.

> While the Matlab server reads the entire array from the dataset and then applies the sub-sampling information, *many* data set types provide ways to subsample variables through their own API (e.g., HDF, NetCDF, . . .). In such cases, you'd read the CE information, then use it to read the data values. The most important point is that *you don't always have to read the entire variable from the data when using the DAP*. In fact, most servers don't, they make sure to use the data set's underlying API in the most efficient way possible, something that the DAP was designed to make possible.

Since the Matlab server supports only Matlab 5 and since Matlab 5 supported only two dimensional matrices, we grab a pointer to the first and second dimensions using the `Array::first_dim()` and `Array::next_dim()` methods. The `Array::dimension_start()`, `dimension_stride()` and `dimension_stop()` methods are used to the read the start and stop indices and the sub-sampling stride for the dimension referenced by the `Pix p`.

```
Pix p = first_dim();
int start = dimension_start(p,true);
int stride = dimension_stride(p, true);
int stop = dimension_stop(p, true);

next_dim(p);
int start_p = dimension_start(p,true);
int stride_p = dimension_stride(p, true);
int stop_p = dimension_stop(p, true);

if(start+stop+stride == 0){ //default rows
    start = 0;
    stride = 1;
    stop = mxGetM(mp)-1;
}
if(start_p+stop_p+stride_p == 0){ //default columns
    start_p = 0;
    stride_p = 1;
    stop_p = mxGetN(mp)-1;
}
```

Using the information from the CE, the array values are sub-sampled and copied to new storage. Again, this step is generally not necessary when it's possible to subsample variables in the data set using an API, et cetera.

```
int Len = (((stop-start)/stride)+1)*(((stop_p-start_p)/stride_p)+1);

int Tcount = 0;
dods_float64 *BufFlt64 = new dods_float64 [Len];

for (int row = start; row <= stop; row += stride) {
    for(int column = start_p; column <= stop_p; column+=stride_p) {
        *(BufFlt64+Tcount) = (dods_float64) *(DataPtr+row+column*mxGetM(mp));
        Tcount++;
    }
}
```

The values, now read and sub-sampled are copied into the DAP variable object. The DAP methods enforce a strict policy that all memory allocated outside of the library must be deleted outside the library, and vice versa. So values sorted in the `BufFlt64` array are copied to new storage allocated inside the `Array` object. This code deletes the `BufFlt64` array.

Also important is the call to `set_read_p()` with the value `true` this sets the `read_p` property[4] so that, should this function be run again while building this particular response, it will know the data have already been read.

```
val2buf((void *)BufFlt64);
delete [] BufFlt64;
set_read_p(true);
```

The remaining software frees resources allocated via the Matlab data access API. As was explained earlier, the `false` return value is a hold over from an earlier version of the DAP library.

```
mxFreeMatrix(mp);
matClose(fp);
return false;
```

[4]The the 'p' stands for 'Predicate'; maybe not the best naming convention given the prevalence of pointers in $C++$ software.

To build a `main()` function that will return the DataDDS, copy the one for the the DDS but change the call for `DODSFilter::send_dds()` to `DODSFilter::send_data()`. This will work because the code leading up to the `send_data()` call builds the DDS object, then the `send_data()` call will arrange to build the DataDDS response using the DDS. During this process it will parse and evaluate the CE and call the `read()` methods for the variables in the DDS. Thus the DDS will contain variables loaded with values which can then be used to create the information in the DataDDS object/response.

10 Notes

Here's a collection of information that might be important to specific servers but is hard to fit into a general tutorial.

- It's possible to replace the three separate programs, one for the DAS, DDS and DataDDS, with a single program. Clearly there all share close to 90% of the same code. You can tell which object is being requested by either using the old Unix trick of making the programs symbolic links to a single executable and then checking to see which name was used to invoke it, or by modifying the dispatch script slightly to pass that information along with the other parameters.
- Similarly, you can subclass the DAS and DDS objects adding methods to them to build up the attributes and variables instead of writing functions to do that task.
- For complex data sets, it's likely that you'll need to create some new fields in the data type classes you specialized.
- If you're writing a server that will principally be for internal use, or you expect many many small accesses, you should look into using the Java software or implementing your server as an Apache module. Contact support or the dods-tech email list for help with the latter.
- You can get help from the DODS web site, the dods-tech email list and the DODS user support desk.
- The DODS Project has developed two tools to help with serving datasets that contain many files. The first is to set up a 'file server' a kind of catalog of URLs that is itself a DODS data set. The second is called the Aggregation Server (AS). The AS can automatically aggregate discrete datasets, accessed as either as files (in some cases) or URLs to produce a single data set. See the DODS web page and/or contact User Support for help with this.
- When you're returning error messages, be sure to trim file pathnames so that the real path to data is hidden.
- If you have a data set and its API requires you to store a lot of state information that's not specific to a given variable or data type, consider creating a mixin class and making your data type children inherit from it in addition to the DAP classes. Because the 'virtual contractors' are used to create instances of the data type classes, you can be sure that in your server, all instances of Byte, etc., contain the mixin. Use the `dynamic_cast` operator to access the mixin's methods.