DODS—Data Delivery Design

23 August 1996

Abstract

This paper describes the design of the HTTP-based data servers for the Distributed Oceanographic Data System, and the clients that access them. The architecture for this system is described in "DODS—Data Delivery Architecture[1]" and the communications protocol used by the components is described in "DODS—Data Access Protocol[2]". Those papers are required reading for this one. The data servers are built by assembling three programs which read parameters passed from National Center for Supercomputing Applications's HTTPD[3] to the programs via the Common Gateway Interface[4]interface. In the design, HTTPD is used as a network I/O management tool which simplifies setup and maintenance of the data server. The programs which comprise a data server return Multipurpose Internet Mail Extensions documents as query results. This paper also describes how to build DODS client libraries which use the information provided by data servers to mimic the responses produced by various data-access Application Programmer Interfaces (API) when they are used to read data files or data sets. While the construction of the data servers is typically straight forward, building the surrogate library for a particular API can be very complicated. This complexity is the result of the need to translate from the DODS Data Access Protocol (DAP) representation of information to the representation used by the API. To accomplish this translation, the programmer must have a solid command of both the DAP and the target API. Although hard to implement, this system has the advantage of providing for interoperability between several different APIs supported by DODS. As new APIs are added the set of interoperable APIs will further expand.

Note

This is a working document made available to solicit comments. To receive e-mail notice of new or significantly changed documents, send e-mail to majordomo@unidata.ucar.edu with 'subscribe dods' in the body of the message.

[1] http://www.unidata.ucar.edu/packages/dods/archive/design/data-delivery-arch/data-delivery-arch.html
[2] http://www.unidata.ucar.edu/packages/dods/archive/design/api/api.html
[3] http://hoohoo.ncsa.uiuc.edu/docs/Overview.html
[4] http://hoohoo.ncsa.uiuc.edu/cgi/overview.html

Hypertext references in the hardcopy version of this document are represented using footnotes that contain a Universal Resource Locator (URL) to the referenced document. To read the online documentation, open the URL 'http://www.unidata.ucar.edu/' with a World Wide Web browser.

This document is written for people interested in implementing software that will be part of, or work in parallel with, the Distributed Oceanographic Data System.

Contents

1 Introduction

This document describes the design and construction of the data delivery components of the Distributed Oceanographic Data System (DODS). The architecture of DODS is described in detail in "DODS—Data Delivery Architecture[5]". DODS is a client-server system which differs from the conventional notion of such a system, such as `ftp`, because it has many different clients, not just one. Instead of the implementor(s) of DODS building a single client program to read and display data, they build replacements for existing data access application programmer interfaces (API). These replacement, or surrogate, API implementations are then linked to any data access, display or analysis software which uses that API to access data. The surrogate API implementations use the DODS tool kit, in conjunction with the HTTPD[6], to access data stored on remote computers which have installed DODS data servers. Because the reimplemented APIs can access data over the network using DODS data servers, application programs linked with them can also access such data, even though the API calls they make are unchanged.

In addition to the remote access of data sets, DODS provides a limited degree of API transparency. The DODS data servers transmit information using the Data Access Protocol (DAP). The DODS surrogate libraries issue requests and read data using the DAP, while the DODS data servers answer requests and provide information using the DAP as well. Both the surrogate libraries and the data servers can be said to *translate* the native access mode of a API or storage format into or out of the DAP. Because a single transmission/access protocol is used to move all information within DODS, any surrogate library can, in principle, access data from any of the data servers, regardless of the storage format of the data accessed by that server. Thus, for example, data stored in NetCDF files can be read using software designed to work with the JGOFS system.

Using the data access protocol has a price. Servers must translate calls in the DAP to calls in the data set API or, if the data set has no associated API, to various reads which get information from the data set. Either way the data server must translate the DAP accesses into the appropriate accesses for the data set. Section 2 describes the design and construction of these data servers. In addition, the surrogate libraries must translate calls from the API they replace to calls in the DAP. Section 3 describes how these surrogate libraries are built.

Several designs were considered for the data delivery mechanism of DODS. They were socket-based peer-to-peer communications, RPC-based peer-to-peer communications, virtual file systems and HTTP/CGI-based client/server systems. The first three of these different designs are compared in: Report on the First Workshop for the Distributed Oceanographic Data System, Proposed System Architecturesand "DODS—Data Delivery[7]", which presents our rationale behind prototyping the RPC-based

[5] http://www.unidata.ucar.edu/packages/dods/archive/design/data-delivery-arch/data-delivery-arch.html

[6] http://hoohoo.ncsa.uiuc.edu/docs/Overview.html

[7] http://lake.mit.edu/dods-dir/dods-dd.html

design for DODS. However, as a result of those prototypes and the development of HTTP as a *de facto* data communications standard, we changed the data delivery design to an HTTP/CGI-based system.

By using HTTP as a transport protocol, we are able to tap into a large base of existing software which will likely evolve along with the Internet as a whole. Because the development of large-scale distributed systems is relatively new[8] there are many problems which must still be addressed for these systems to be robust. These problems include naming resources independently of their physical location, and choosing between two objects which appear to be the same but which differ in terms of quality. These are general problems which are hard to solve because they will be solved effectively only when the Internet community reaches a consensus on which of the available solutions are best. HTTP, because it is so widely accepted, provides a reasonable base for such solutions. This view is supported by the recent Internet Engineering Task Force (IETF) work on extending the HyperText Transfer Protocol[9] and HyperText Markup Language[10] standards.

1.1 Process Configuration

The collection of processes involved in a single data transfer from a remote archive to a user program is shown in Figure 1. In the figure, the components of each process are also shown. Initially the user program, which has been re-linked with the surrogate API implementation[11], sends a Uniform Resource Locator (URL) to the HTTPD[12]. HTTPD decodes the URL and executes the named CGI, passing to it parameters via environment variables. The CGI is run in a separate process, so once started, it operates independently from HTTPD. The CGI in turn examines its arguments and determines which of the DODS filter programs to execute. The filters write to standard output which, in this case, is a socket connecting the filter to the remote client (because the filter is a child process of the CGI which in turn is a child of `httpd`; `httpd`'s standard input and output are connected to the remote client).

Each machine that makes data available to DODS clients must, first and foremost, have all the data to be accessible *on line*. By on line we mean that the data must be accessible without human intervention and without significant delay. It is unreasonable to expect that data to be incorporated into a user's display package will take a day to be transferred to their computer! Ideally, DODS users will not be able to tell that data they access is remote. This does not mean that data must be on a

[8]Even though the UNIX/Internet combination has existed for more than 10 years, it is only recently that systems like the WWW have seen widespread use. Previously data was shared over the Internet using ftp or similar systems.

[9]http://www.w3.org/hypertext/WWW/Protocols/HTTP/HTTP2.html

[10]http://www.w3.org/hypertext/WWW/MarkUp/MarkUp.html

[11]In order to use a given data display or analysis program with DODS, users or system managers must re-link those programs with the DODS reimplementations of the APIs used by those programs. DODS currently supports two APIs: netCDF and JGOFS. Support is planed for HDF and the formats GRIB and BUFR.

[12]http://hoohoo.ncsa.uiuc.edu/docs/Overview.html

hard disk, however. Data could be stored on an auto changer, in a special purpose database machine or other mass storage device.

In addition to an on line data set, data providers will need to run HTTPD and will need to install a CGI and a set of filter programs. HTTPD is used as a network interfaceq, security and access logging tool for DODS. It is a standard piece of software distributed by National Center for Supercomputing Applications and is already installed on many computers.

In addition to HTTPD, a DODS data server consists of a CGI and a least three filter programs (See Figure 2). The three filters handle, respectively, the client's requests to get the Data set Attribute Structure (DAS), the Data set Description Structure (DDS) and to read the values of variables in the data set. These three executables along with HTTPD and the dispatch CGI make up a complete DODS data server.

1.2 Client/Server and Program/Library Interfaces

The DODS client library and data server programs communicate information using URLs and MIME documents. Figures 3 and 4 show these two communication paths.

All information sent from the filter programs to the client library is enclosed in a MIME document. Two of the three programs return information about the variables contained in the data set as `text/plain` MIME documents. These documents can then be parsed by software in the client library. In addition, these text documents can be read by any software that can process `ASCII` text. Thus, the responses made by the server are specifically suited to use by the DODS client libraries, they can also be used by many more general programs. For example, it is possible the use a general purpose World Wide Web browser to 'read' these documents.

The third filter program returns binary data encoded using Sun Microsystems External Data representation (XDR) scheme prefixed with text which descibes the data type of the binary information[13]. The data is enclosed in a binary MIME document whose subtype is experimental. This document can be read by software that is part of the client library using additional information about the data set that is stored as state information by the client library. Because correct interpretation of this document requires knowledge of the format of the binary information, it is not possible for general purpose WWW browsers to interpret this file (although most browsers can read and save to disk any arbitrary data).

In order to provide link-time compatibility with the original API libraries, the DODS client libraries must present *exactly* the same external interface as the original libraries. However, these new libraries perform very different operations on the data (although, for a API used to access a self-describing data format the operations are analogous). One difference between the two is that most data access APIs use file names to refer to data sets. In the simplest case these file names are given on the command line by the user and passed, without modification, to the API. The API

[13]The text is a mini-DDS which can be parsed and instantiated using the DODS tool kit software.

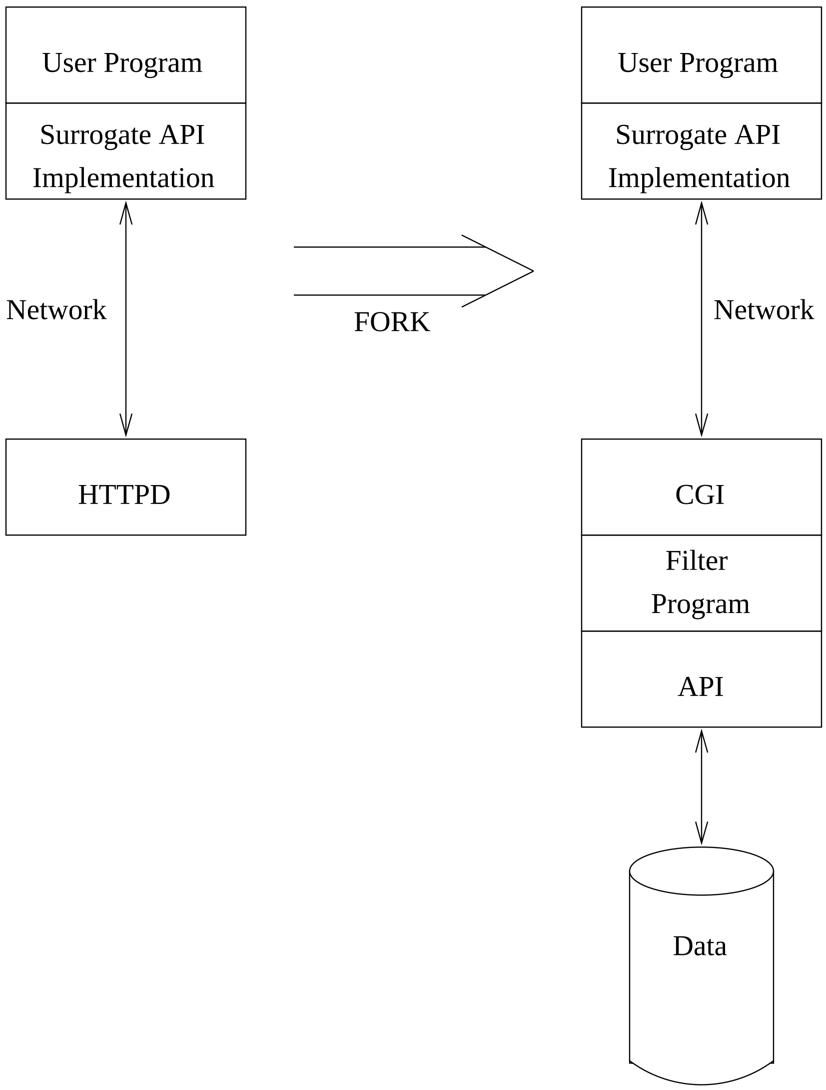

Figure 1: Run-Time Processes. The client connects to a HTTP server which then forks and executes the CGI/Translation module software. The latter is used to access the data.

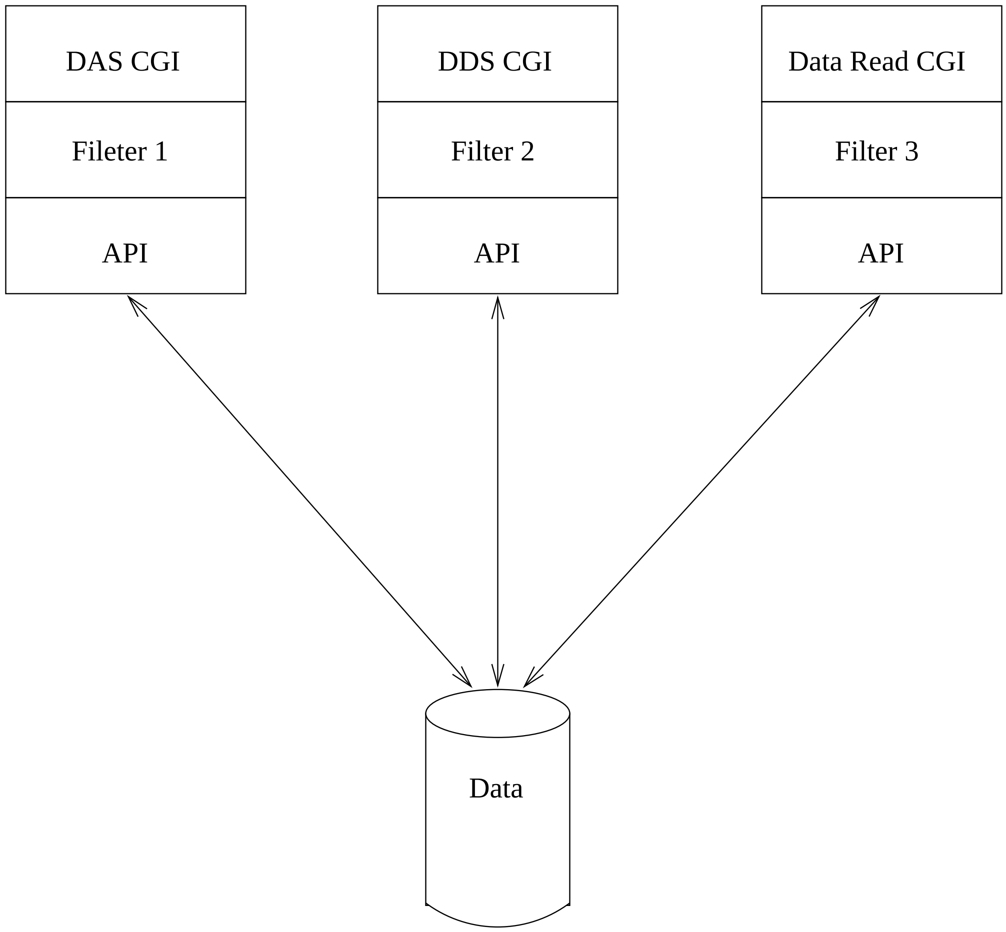

Figure 2: Executables needed to support access to a data set for DODS clients. This collection of three CGIs along with the appropriately coded translation modules and linked to the original implementation of the API used to access the data set constitutes a *Data Server* for DODS.

uses the file name to open a file and returns an identifier of some type to the user program. Subsequent access to the data are made through this identifier.

In this simple scenario, it is possible to substitute a URL in place of a file name (in part because both are stored in string-type variables). This same user program can be invoked, on the command line, using a URL in place of the file name. The program will, in almost all cases, pass the URL to the API to open the data set. However, since the user program has been re-linked with the DODS re-implementation of the API, the functions in the API will correctly interpret the URL as a remote reference (see Section 3.1.1). Clearly, one requirement that a user program must meet in order to be re-linked with DODS is that it must not itself try to open or otherwise manipulate the 'file name' which will be passed to the API[14].

As shown in Figure 4, the client library does manipulate the URL used to access the data set. A DODS data server is actually comprised of three programs which each respond to a different type of data as described by the Data Access Protocol. These programs are accessed as WWW documents via the HTTPD using the CGI mechanism. By *convention* the data set name consists of the HTTP access string (`http://`) followed by the machine name and a string which the data providers choose depending on the name and location of the dispatch CGI and the data. In order to read the DAS, DDS or data, this URL is modified, again by convention, by adding a suffix. The three suffixes currently (as of 23 August 1996) recognized are `.das` for the dataset attribute structure, `.dds` for the dataset descriptor structure, and `.dods` for binary data.

2 Data Servers

A DODS data server is a collection of three executable programs and a CGI which provide access to data using HTTP. Each of the program/CGI units is capable of satisfying one of the three requests which are defined in the Data Access Protocol. The DAP defines two ways to access metadata which describes the contents of the data set (one for use by the DODS surrogate libraries and one for use by third-party software and users) and one way to access data. Data access is accomplished by reading the variables which comprise the data set. This access can be modified using a *constraint expression* so that only portions of the variable are actually read from the data set.

An essential characteristic of a DODS data server is that it be capable of evaluating the constraint expression associated with a data access request. In many cases reading a single variable from a data set results in data which is of little or no interest to the user. Often users are interested in those values of a variable which meet some additional criteria (e.g., they fall within a certain time range). For a complete description of the data types supported by DODS and the constraint expression operators,

[14] It is possible that a user program might perform an operation such as looking in one data base for a file name and then opening the named file—this doesn't count as a manipulation of the file name.

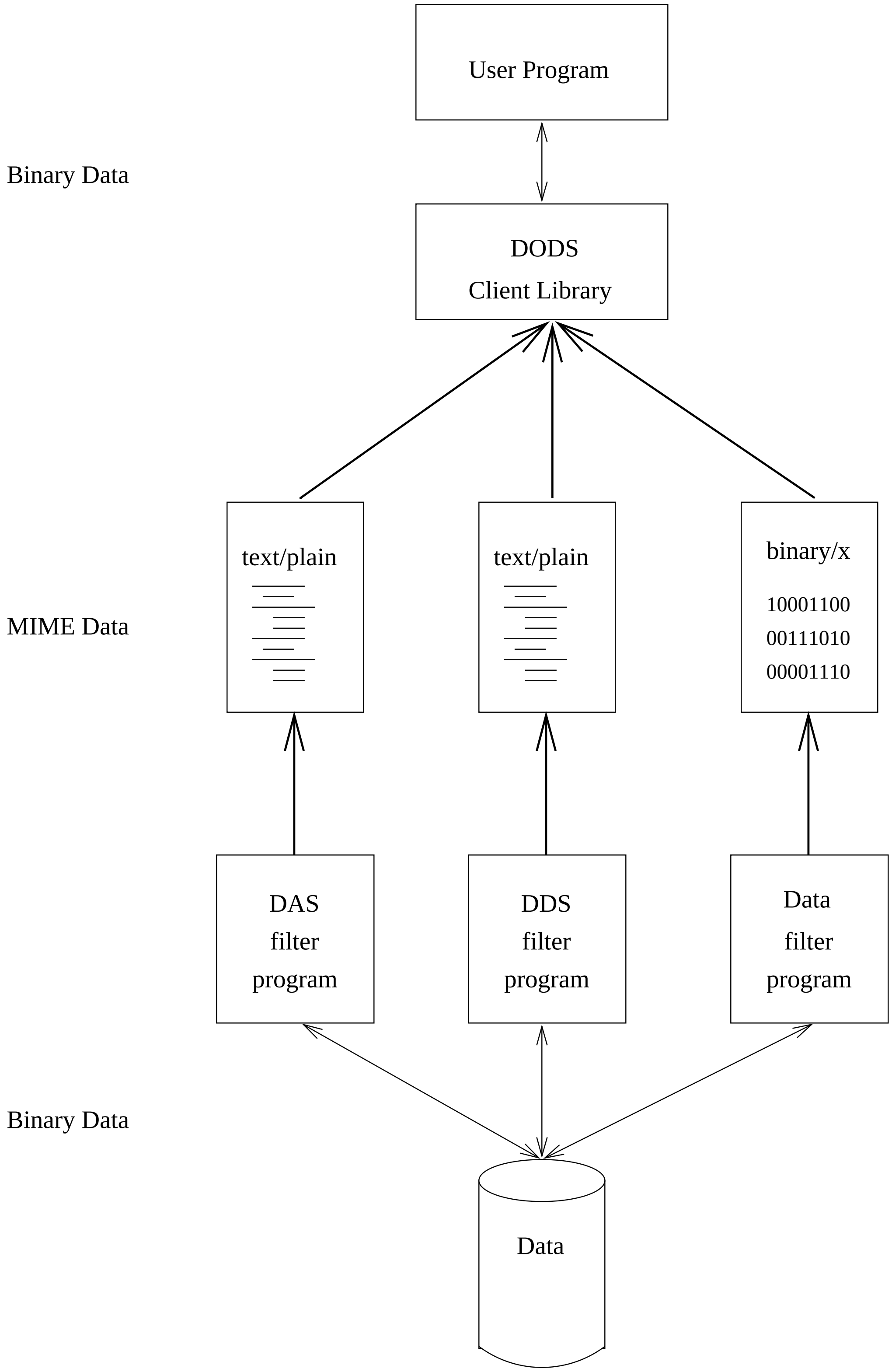

Figure 3: MIME documents are used by the server programs to return information to the client processes.

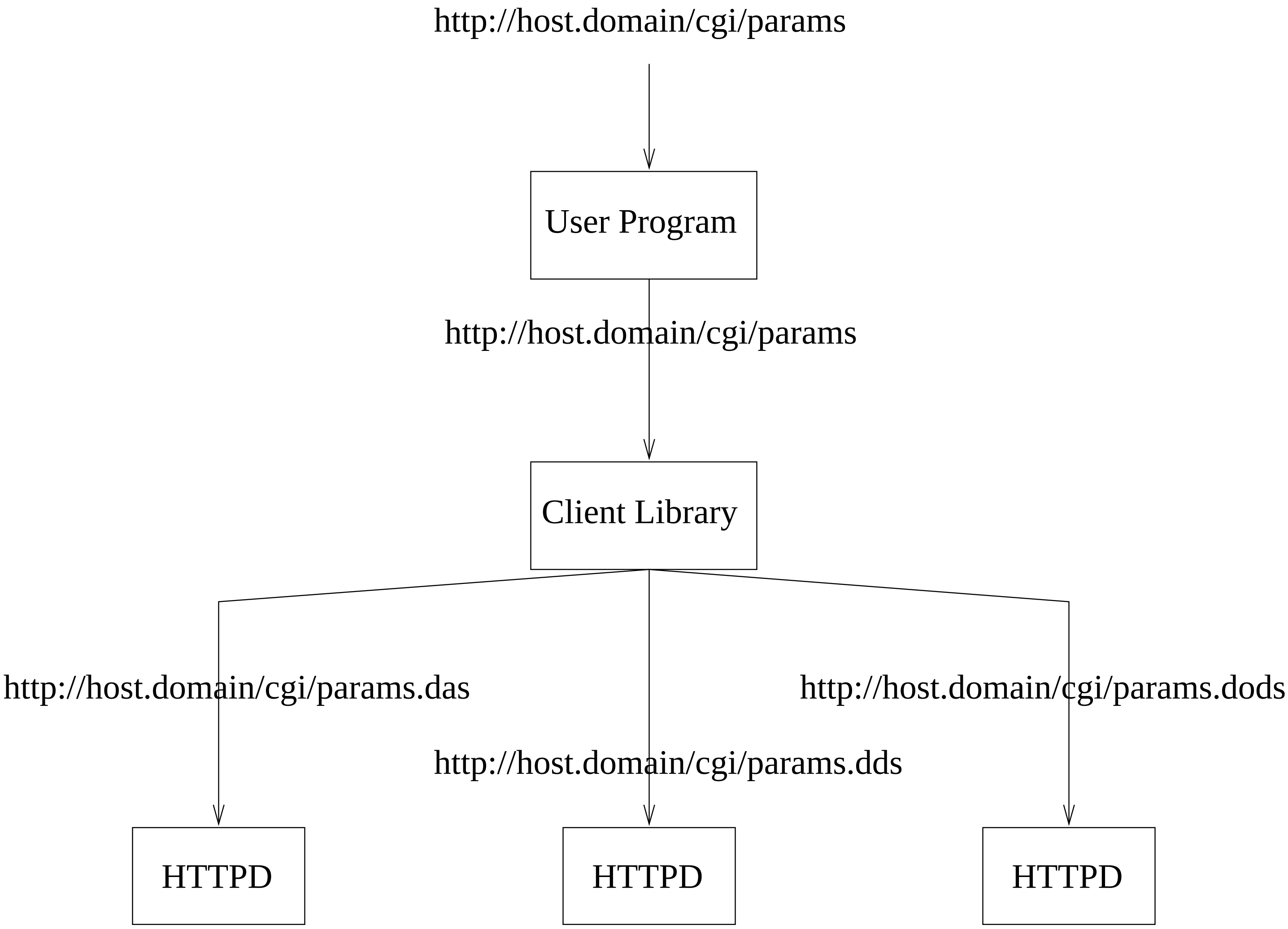

Figure 4: URLs are used in place of filenames to reference remote data sets accessible via a DODS data server.

see "DODS—Data Access Protocol[15]".

2.1 Basic Requirements for the Data Servers

The data servers must satisfy two requirements: they must provide access to data via the DAP and they must use on-line data without requiring its modification.

Providing access to data using the DAP is necessary because that is how the DODS architecture provides interoperability between different APIs. Because the data servers translate accesses to a data set from the DAP into either an API (e.g., NetCDF if the data set is stored using that API) or a special format (e.g., GRIB), any (client) process that uses the DAP can access the data. The underlying access mechanism is hidden from the client.

In the current design of DODS, meeting this requirement means that for each API or format in which data is stored, a new DODS data server must be built[16].

The other requirement which each server must satisfy is that data, however it is stored, should not require modification to be served by a DODS data server. This is important because many data sets are large and thus very expensive to modify. It is a poor practice to force data providers to modify their data to suit the needs of a system. Rather, DODS data servers must be able to translate access via the DAP into the local storage mechanism without changing that local storage mechanism.

This requirement limits DODS to those APIs and formats which are, to some extent, self-describing. Because the DAP bases access on reading a named variable, it must be possible, for each data set, to define the set of variables and to 'read' those variables from the data set. However some data sets do not contain enough information to make remote access a reality. Instead, additional information, not in the data set itself, is needed. This information can be stored in ancillary data files which accompany the data set. Note that these files are separate from the data set, they are not added to the data set and do not require any modification of the data set.

2.2 Construction of the Data Server

The data server structure is shown in Figure 2. The 'data server' is actually a collection of three separate programs which correspond to the three data objects in the DAP. There is no coordination of effort between the three programs; each operates independently[17].

Each of the three programs which comprise the data server use the Common Gateway Interface[18]mechanism to receive parameters from HTTPD[19] (which is used

[15] http://www.unidata.ucar.edu/packages/dods/archive/design/api/api.html

[16] In practice data servers for the supported APIs and several data formats will be available with the DODS software distribution.

[17] Some optimizations may introduce limited dependence via a data cache.

[18] http://hoohoo.ncsa.uiuc.edu/cgi/overview.html

[19] http://hoohoo.ncsa.uiuc.edu/docs/Overview.html

to invoke the programs and manage the network I/O connections—see Figure 1). Once parameters have been parsed, the programs read information from the data set which is used to satisfy the query. Accesses to the data set are made using an established API. Thus each of the programs consists of two modules:q CGI interface and a data-access API. The paper "DODS—Client and Server Toolkit[20]" contains information on software which is designed to aid in the construction of these parts of the data server software.

2.2.1 Building the DAS and DDS filter programs

The three programs which comprise a DODS data server can be speparated into two groups, one which provides access to 'metadata' and one which reads data variables from the data set. The DAS and DDS are used by the DAP to transfer information about the data set to the client. As described in "DODS—Data Delivery Architecture[21]", the client must decode these structures and use some of the information in orchestrating variable reads. While the DAS and DDS contain information that is semantically distinct, the process of building these two structures is almost identical.

Figure 5 shows a structure chart for the modules that build the DAS[22]. In order to build the DAS object, the DAS program must parse the parameters passed to the CGI. Once the parameters have been parsed, the program must build a DAS object. The construction of the DAS object can be simplified by using the DODS Client and Server Toolkit software. This software handles the reading data set attribute information stored in the ancillary files (see the DAP section on ancillary data) and integrating it with information that is already part of the DAS object. The toolkit also contains contains software which builds an ASCII text representation of the DAS for transmission to the client.

In order to completely build the DAS it is, in general, necessary to read the data set using whatever mechanism is natural for those operations. In many cases, data sets which are part of DODS will be accessible using an established API. Thus, part of the DAS data server program must use the data set's API to read information about the variable's attributes (as defined by the data access protocol) and record them in the DAS object. The client and server toolkit software contains DAS object member functions which simplify this task somewhat, but people who intend to build the DAS program must learn both what type of information is stored in the DAS object as well as how to read that information using the data set's API.

The construction of the DDS is essentially similar to that of the DAS. The real difference between these two programs is in the content of the object which they

[20] http://www.unidata.ucar.edu/packages/dods/archive/implementation/toolkits/toolkits.html

[21] http://www.unidata.ucar.edu/packages/dods/archive/design/data-delivery-arch/data-delivery-arch.html

[22] Structure charts are used to describe the design of the data server programs rather than object diagrams because these components depend, to a large extent, on the functional APIs used to access the data sets.

build and send to the client. Thus, the only functional difference is in the specific API function calls used to build the object.

2.2.2 Building the Data Read Program

While the two programs used to send the DAS and DDS to the client both may read from the data set, the *send data* program necessarily *must* read from the data set. This program reads the values of a single variable in the data set (as defined by the Data Access Protocol) and packages it in a Multipurpose Internet Mail Extensionsdocument. This MIME document is then sent to the client process. The client process must decode the binary MIME document and internalize the data it contains (see Section 3).

A structure chart for the send data program is shown in Figure 6. This structure chart shares some features with the DAS chart in Figure 5; it also has modules for CGI parameter parsing and construction of a MIME document. However, there are some important differences between the two figures. The MIME document sent to the client by the send data program contains both text and binary parts, while the DAS and DDS programs send text-onlyq MIME documents. In addition, the send data program uses the DDS object (which contains instances of objects for each variable in the data set) when it performs the read operation.

The implicit dependency between the DDS filter and data filter programs is important. In order for the send data program to function, it must invoke the read member function for the requested variable. However, to get the variable object, the send data program must have access to the DDS object. Thus, before send data can do anything beyond parsing its arguments, it must build the DDS object. This may be accomplished in several ways: the DDS object may be cached (either as binary data or using the text representation built for transmission to the client), it may be built explicitly by the send data program via the DDS program (the send data program acts as a client of the DDS send server program), or it may be built by including the DDS send program's logic in the send data program.

Once the DDS object has been instantiated by the send data program, it must invoke a member function to read the desired variable from the data set. The read member function for each class of variable described in the data access protocol must be specialized for any API used to read the data set. The specialization of the data type hierarchy for a particular API is described in the documentation for the client and server toolkit.

Once the data for the variable has been read into the correct object within the DDS, the data filter must build a multipart MIME document so that the data can be returned to client program. This MIME document contains a text part and a binary part. The text part consists of a DDS which describes just the variable in this document (as opposed to every variable in the entire dataset) This is straightforward to make given the completet DDS and the client and server toolkit software. Following the text section, the data filter program must append the value of the variable (encoded using XDR).

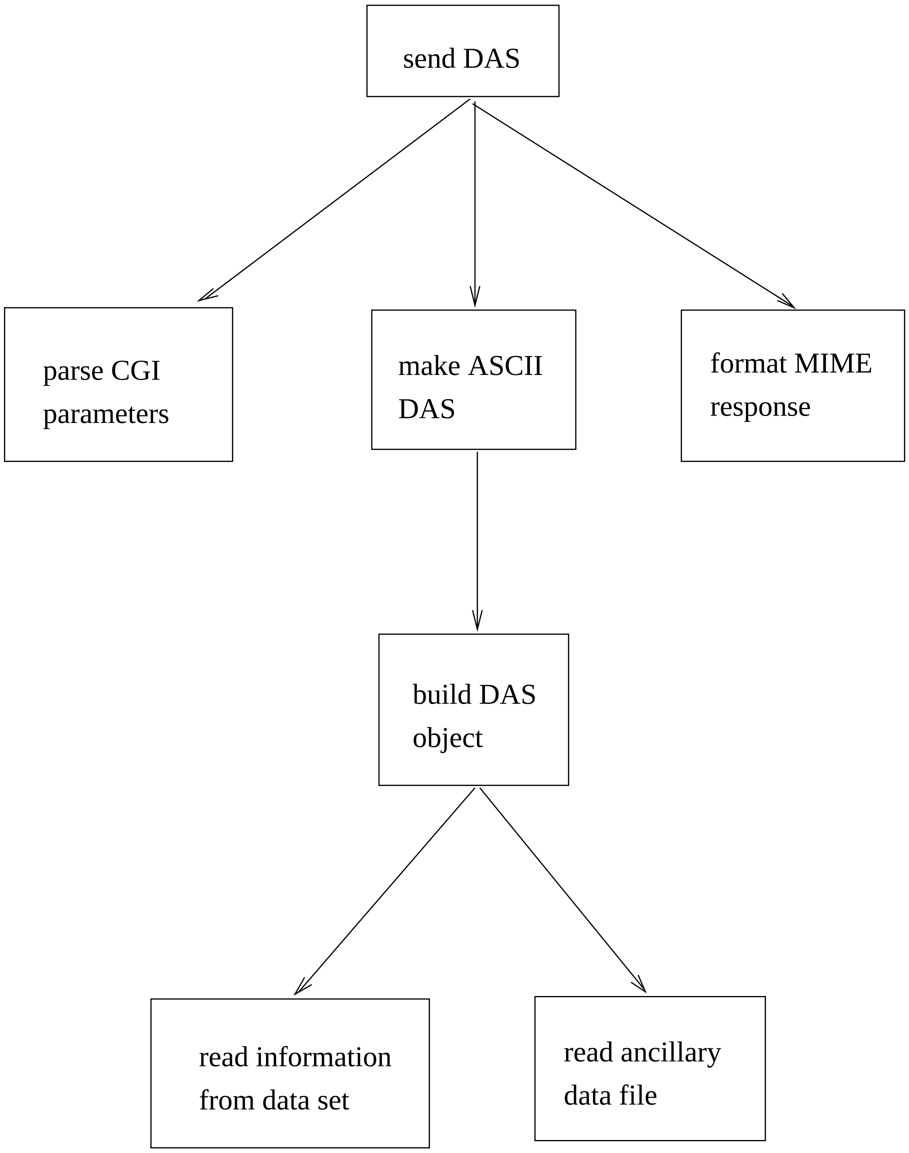

Figure 5: Structure Chart for the Data set Attribute Structure Server Program.

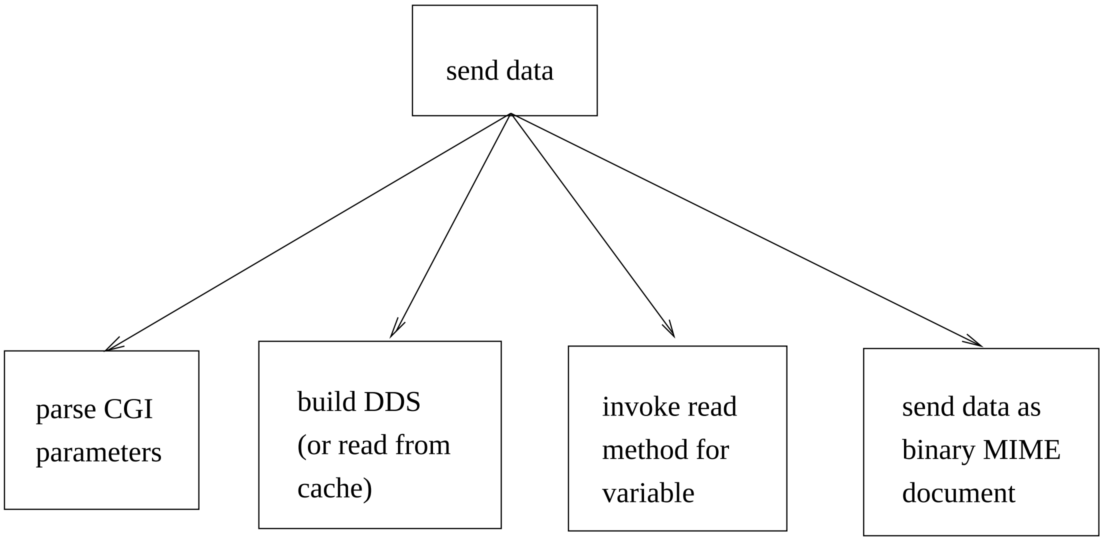

Figure 6: Structure Chart for the Data set Variable Read Server Program.

2.3 Data Server Optimizations

There are two types of optimizations which can clearly boost performance of a DODS data server[23]. Both of these optimizations use caching to store results from one of the programs for later use or reuse by that or another program.

2.3.1 Reusing DAS and DDS Objects

Because the construction of the DAS and DDS objects requires that an entire data set be scanned, it can become very inefficient to continually rebuild these objects. Because the DAS and DDS server programs use a text representation for transmission of this object from the server to the client, and because text is easily stored by computer systems, it is possible to store both the DAS and DDS objects once they have been created. Subsequent accesses to these objects can be accomplished by reading and transmitting the textual representation without, in the case of these two programs, actually building the binary data object.

When taking advantage of this optimization, it is important that the server check the date stamp of the DAS/DDS text objects and compare it to the latest modification date of the data set. For any data set to which new data is periodically added, the DAS/DDS text object must clearly be updated so that the cached text object matches exactly the object that would be created if the object were built by querying the data set.

The update of the DAS/DDS text object can itself be optimized significantly. It is not actually necessary to completely re-read the entire data set. Because the software used to build both the DAS and the DDS binary objects work incrementally, it is

[23]Which is actually a collection of three largely independent programs.

possible to read the DAS/DDS text object into memory (i.e., build the binary object from its text representation) and then read only the new parts of the data set. The binary object will be added to as needed. NB: The DAS/DDS software may not properly update *changed* data (data that was present in the previous incarnation of the data set, but is now different in the current version) nor is it straightforward to remove data which is no longer present in the data set.

2.3.2 Using the DDS object by the Send Data Program

Once the data server programs have been optimized to store the DAS and DDS text objects for future use, it is a simple matter to modify the send data program so that the cached DDS text object is used by that program.

3 User Programs and DODS Client Libraries

Most client/server systems are built around a fixed set of client and server programs. Often, the designers of such a system specify a communication protocol to be used by the two processes and then design a server and a client program. Because the communication protocol used by the system is documented, it is fairly easy for a third party to build new clients (or servers, although that is less common) for the system. An example of such a system is the World Wide Web designed initially at the European Laboratory for Particle Physics. Subsequent to the Web's appearance, many new client programs have been introduced by third parties. In fact, the DODS client libraries are a type of WWW client program.

However, the DODS client libraries are different from most WWW browsers because they are themselves used to build programs. The DODS netCDF library, for example, can be used to add network connectivity to any program which uses netCDF. Even though the DODS client libraries contain all the software needed to be full-fledged WWW clients, they are *not* intended to be stand alone programs themselves; instead they are tools which can be used to add network I/O capability to programs which already use one of the DODS supported APIs.

Because the DODS client libraries are designed to be complete *replacements* for existing libraries, they must be written so as to faithfully emulate the original implementation's behavior. If the client library meets this criterion, then it can be used with existing programs without requiring those programs be modified in any way. This is the fundamental requirement which any DODS client library must meet.

3.1 Client Library Functions

The central premise of the DODS data delivery architecture is that it is possible to categorize many APIs used to store/access scientific data as following closely with the UNIX file system paradigm of open, read and close[24]. Furthermore, it is assumed that any state information used by the API can be stored locally in the client process and does not need to be retained by the data server.

In order to build the client library for a particular API, it is necessary to re-implement that API so that it uses data that can be acquired from the DODS data access protocol rather than the files or other data store it was originally design to use. The client library must also maintain the illusion of a connection (state) for each open data object (typically a file) even though no such connection actually exists (because the DAP is a stateless protocol).

To accomplish these tasks with a minimum of difficulty, it is useful to divide the API to be re-implemented in five types of functions:

- Open or connect
- Variable information read
- Data read
- Write
- Close or disconnect

3.1.1 Rewriting the Open and Close Functions

The functions which perform the open/close operations will require complete re-implementation so that information about the data set can be retrieved from the data server. These re-implemented functions must store the necessary state information so that subsequent accesses for variable information or data reads can be satisfied. This state information will, in almost every case, consist of the data set attribute structure (DAS) and dataset descriptor structure (DDS).

Figures 7 and 8 are structure charts for the DODS client library open and close functions. In order to write the DODS client library version of *open* for a given API, the function must first determine if the data object (typically a file) is local to the user program making the open call or is a remote data object to be accessed through DODS. It is possible to access DODS objects which are local to a user program, but there is little reason to do so if the data object can be accessed through the API used by the user program. In any case, the distinction of local or remote is made on the basis whether a URL is used to reference the data object. If that is the case, the object is assumed to be remote and is accessed using a DODS data server, otherwise it is assumed to be local and is accessed using the functions of the original API implementation.

If the data object is remote, then the open function must build a structure which can hold the DAS and DDS objects which describe the named data set. Once this

[24]Because the DAP is a read-only protocol, we are not concerned with any operations that write data.

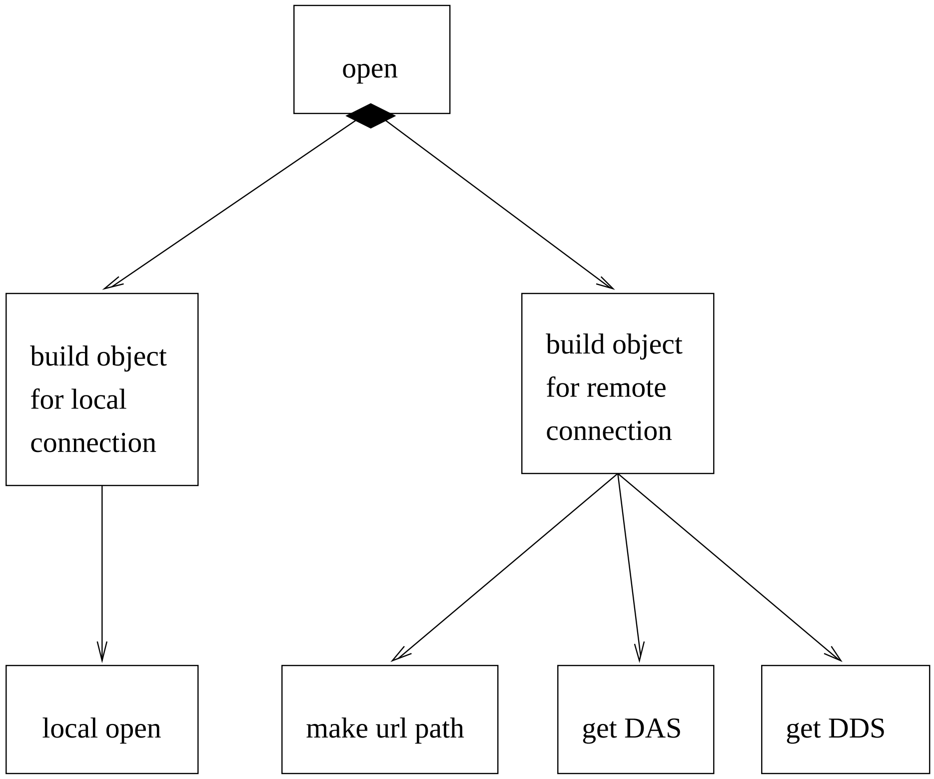

Figure 7: Structure Chart for the Open/Connect Functions in a DODS Client Library

object is built, the open function must map this structure to a file identifier or pointer which can be passed back to the user program as the return value of the open function. Subsequent accesses to the data set will include this identifier (or pointer), and each function which is a member of the API can be modified to use it to gain access to the state information stored by the open function.

The close function must use the state information accessible with the file identifier or pointer returned by the open function to determine if the data set is local (and hence manipulated using the original API implementation) or remote. In either case, the appropriate actions necessary to free allocated storage, ..., must be taken. In the case of a local data set, the original implementation's *close* function must be called. In the case of a remote data set, the locally stored state information must be freed.

The network I/O tool kit which is part of the DODS software distribution contains utility functions and classes which simplify most of these operations.

3.1.2 Getting Information about Variables

Most APIs for self-describing data sets include functions which return information about the variables that comprise a data set. These functions return information about the type and shape of variables in a form that can be used by a program as well as ancillary information about the variables that is more often than not intended

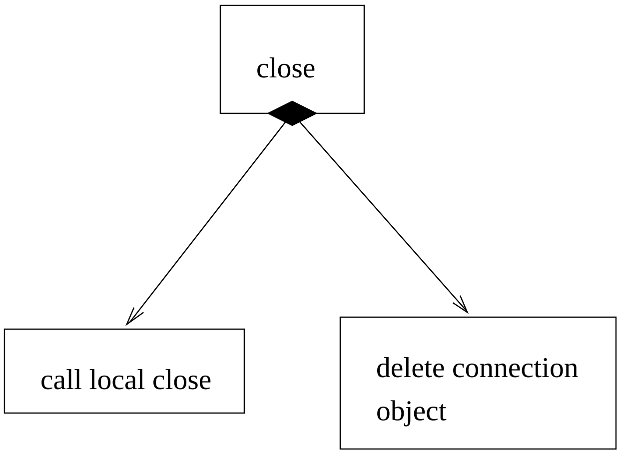

Figure 8: Structure Chart for the Close/Disconnect Functions in a DODS Client Library

for use by humans. Each of these functions must be rewritten so that to the extent possible, information present in the DAS and DDS is used to satisfy them.

Figure 9 shows a simple structure chart for this type of operation. While many 'self-describing' APIs may have 20–30 or more of these functions, the basic structure of the re-implemented code is the same for each one. If the data set is local, use the original implementation, otherwise use the locally storedq state information (DAS and DDS) to answer the function.

However, rewriting these functions can be the most labor intensive part of re-implementing a given API. This is typically the largest group of functions in the API and the information stored in the DAS and DDS must often be 'massaged' before it fulfills the specifications of the API. Thus the rewritten functions must not only get the necessary information from the DAS and DDS objects, but they must also transform the types of the objects used to return that information to the user program into the data types the program expects.

The DODS Client and Server Toolkit contains C++ classes which both define the DAS and DDS objects and provide member functions which simplify this process.

3.1.3 Reading the Values of Variables from a Data Set

Reading variables from a data set is the last type of API function which requires serious attention. In order to accomplish this task, the API functions which read data must be identified and modified to use data gathered in the objects which are part of the DDS object. However, before any of the variable-type objects in the DDS can contain any information a request to read one or more variables must be issued to the remote data server.

In order to read the contents of a variable from the data set into the appropriate object within the DDS, the API read function(s) must be rewritten so that any information needed to build the correct URL is retrieved from the DDS. Once the

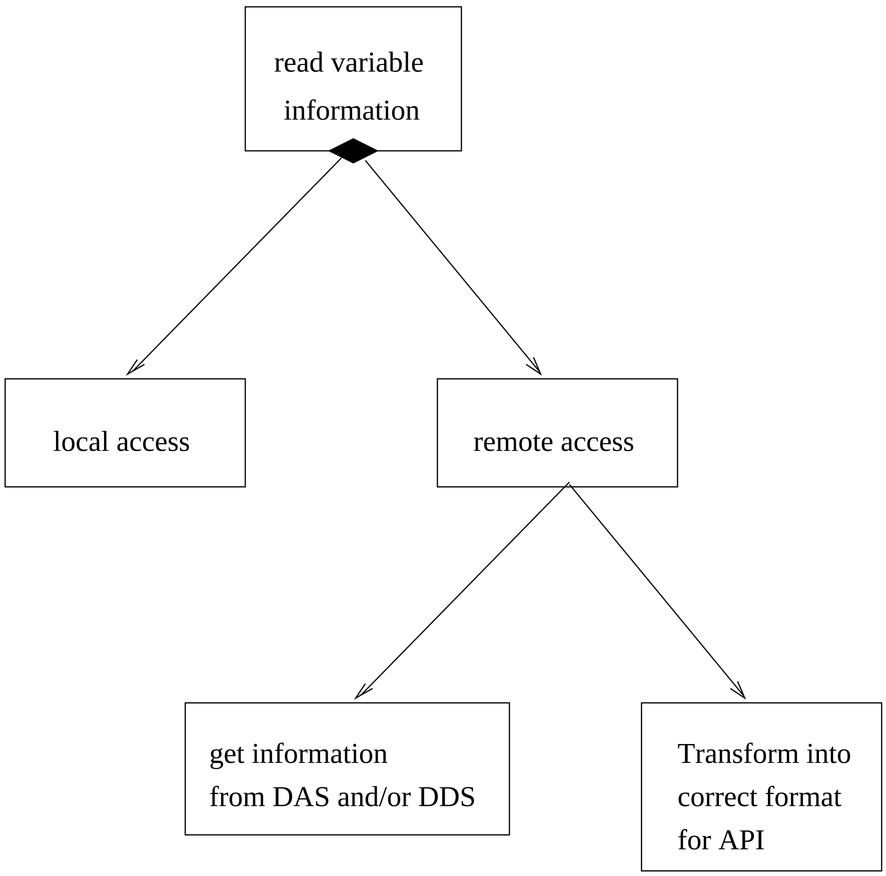

Figure 9: Simple Structure Chart for the Metadata Access Functions in a DODS Client Library

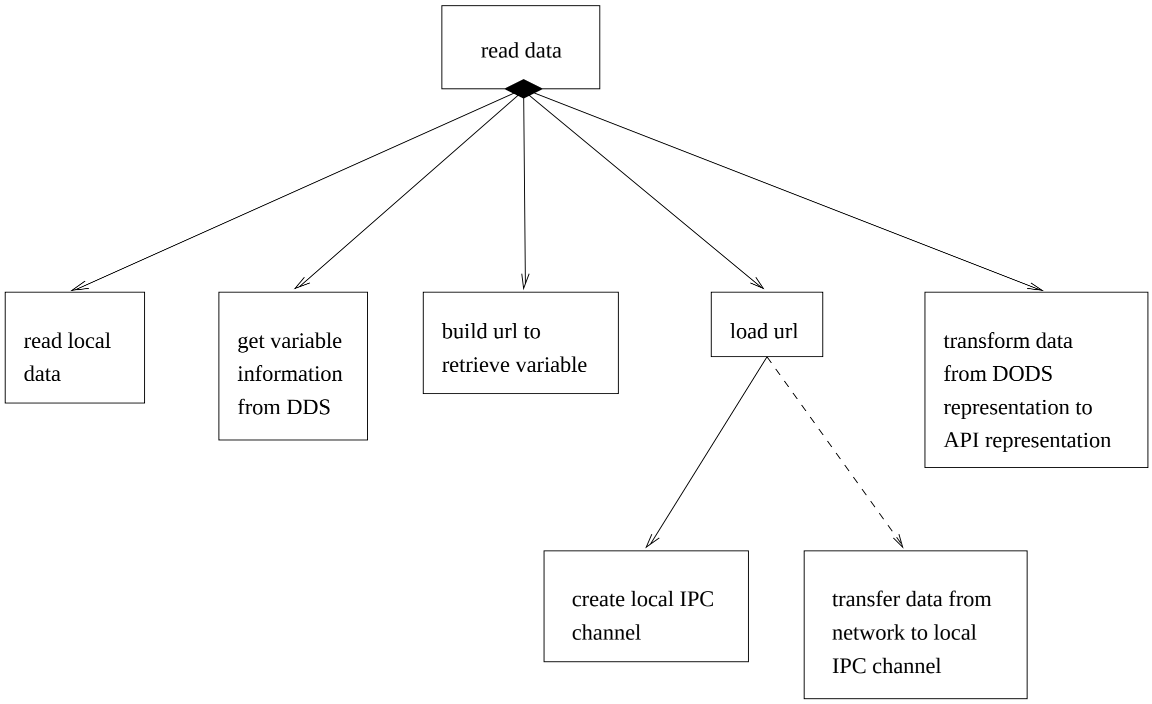

Figure 10: Structure Chart for the Data Read Functions in a DODS Client Library

URL has been built it is used to access a MIME document which contains, in binary form, the data in the variable. This information is then added to the correct object in the DDS. The read function then reads this information and returns it to the calling program.

All binary (i.e., variable value) information transferred by DODS is transfered as multipart Multipurpose Internet Mail Extensions documents. These documents contain a text description of the data type and the binary representation of the variable's value (the later is encoded using Sun's External Data Representation (XDR) standard). Once the binary data is received by the client library, it is transformed, using functions from the XDR library, from XDR's canonical network representation to the local machine's representation. This binary information is then stored in the object with the DDS. Thus, while binary information is transferred from server to client using XDR, the binary information stored in the DDS is in the client CPU's representation.

The DODS client and server toolkit provide C++ software which can be used to simplify reading variables and decoding the XDR-encoded binary data. However, each client library must provide code which extends the client and server tool kit class library to handle any special formatting/typecasting requirements of a particular API.

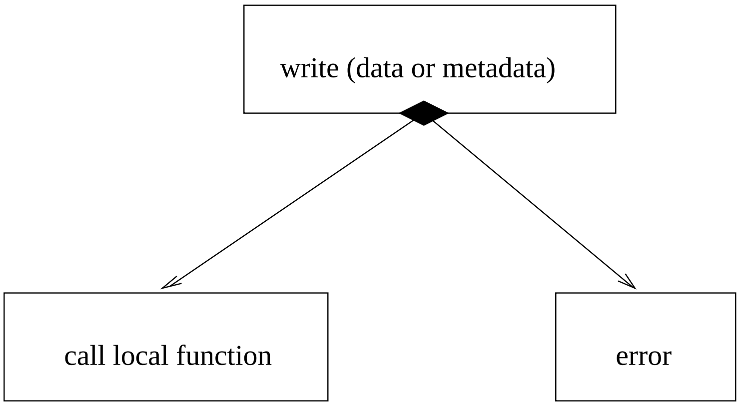

Figure 11: Structure Chart for Functions that Write to Data Sets

3.1.4 Functions that Write to Data Sets

DODS is a read-only data system. While it is not technically inconceivable, a system which allows modification of remote data sets is operationally much more complex. Thus, functions that write data are rewritten so that they call the original implementation in the case of a local access or return an error code in the case of a remote access. The error code should indicate a recoverable error so that programs which perform both reads and writes can recover if their logic permits. Figure 11 shows the structure of the typical write function in-the DODS client library.

3.2 Adding Local Access to a DODS Client Library

In order to ensure that programs, once they have been re-linked with DODS client libraries, can still access local data stores (e.g., files) is is necessary to add software which can read those local data to the functions in the re-implemented library. Typically in each function in the new library the state information accessed by the identifier passed to the function is used to determine if the call is to access local or remote data. In the former case, the function must do exactly what the original implementation of the API would have done to satisfy the function call.

It is wasteful to completely recode the entire API just to achieve local access, however, it is also not possible to simply link the user program with both the DODS client library and the original library. It is not possible to link with both the DODS client library and the original library because both must *define the same external symbols*. Linking with both libraries will produce link-time conflicts on most computers or result in an incorrectly linked binary image.

In order to use the original implementation of the library, it is necessary to rename all of its external symbols that will appear in user programs. For example, if an API defines four functions (open, close, read and write) and one global variable (errno), then each of those must be renamed to some new symbol (e.g., orig_open, orig_close, . . .). These source modules can then be added to the set of object modules used to build the DODS client library. Of course the DODS client library must also include the original external symbol names; one approach is to recode each of the APIs external symbols as a function which either calls the DODS-replacement or the original function (now renamed so that the symbols do not conflict) depending on whether the access is local or remote.

4 Conclusion

This paper describes the design of the data delivery components of the Distributed Oceanographic Data System. This design uses HTTPD and the CGI interface to provide access to remotely stored data. User programs, which were never intentionally written with access to remote data in mind, can be re-linked with DODS client libraries and access data using HTTP. The DODS client libraries are direct call-for-call replacements for existing APIs, thus any user program written to make use of an API for which a DODS client library exists can be re-linked and can access data via the DODS HTTP-base data servers. Because the data servers and client libraries use a single protocol to provide access to the data regardless of the data's storage format, user programs written to read data using one API can actually access data through DODS data servers in several APIs.

The current design of the DODS data delivery mechanism as described here and in "DODS—Data Delivery Architecture[25]" and "DODS—Data Access Protocol[26]" is the result of our previous experience building a similar system using Sun Microsystem's Remote Procedure Call (RPC) technology. That system demonstrated the feasibility of re-linking user programs with re-implemented API libraries and using those re-linked programs to access remotely stored data. That design, however, did not use a single transmission protocol for all servers and libraries. Thus it was not able to provide interoperability between APIs. This design reflects our desire to achieve that goal.

The decision to abandon the RPC-based transfer technology in favor of HTTP-based communications rests on the growing prominence of that technology, the convenience that its supporting software provides and the likelihood that a variety of interfaces can be constructed to access the HTTP-based data servers. Because HTTP is receiving widespread use, there are many active efforts to refine its weak points. By adopting HTTP, DODS stands to benefit from these refinements. In addition, using text MIME documents to transmit two of the three responses from the data servers makes interfacing our data servers with many different programs, in addition to programs re-linked with DODS client libraries, possible. For example, it is possible to use popular WWW browsers to read information about the contents of DODS data servers.

[25]http://www.unidata.ucar.edu/packages/dods/archive/design/data-delivery-arch/data-delivery-arch.html

[26]http://www.unidata.ucar.edu/packages/dods/archive/design/api/api.html