DODS—Data Access Protocol

23 August 1996

Abstract

This document describes a stateless data transmission protocol which can be used to access a wide variety of earth sciences data. The protocol is referred to as the Data Access Protocol in much of this document and it is intended to be used in the construction of system software, particularly for the Distributed Oceanographic Data System. This protocol for data access is flexible enough to access all the data sets which will be part of the system. However, it does not offer the rich features found in interfaces intended for user-program construction. In addition to presenting the requirements and a design for the interface, the rationale for choosing this type of interface is also given.

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.

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

The Distributed Oceanographic Data System (DODS) Data Access Protocol (DAP) is a *lingua franca* for reading earth science data sets. It is used as an intermediate representation for data which are nominally accessed using an established third-party Application Programmer Interface (API) (e.g., netCDF). Because, in addition to a method for data access, the DAP defines the content of ancillary information about data sets, it provides the means to access data sets through a single protocol using any of the DODS supported APIs. Because the data access protocol serves as an intermediate representation for several different APIs, it can be used to translate between any two of those APIs.

In DODS—Data Delivery Architecture[1], an architecture is described that uses third party APIs to access data stored on remote computers. In that document, it is assumed that users of those data write programs in which those data are accessed using an API. For DODS, this assumption is advantageous because it limits the class of programs which DODS must address to those which use a formally defined interface for data access. For the user, it is advantageous to use an API because the storage format of the data is both hidden from the user and separated from the user program source code, resulting in programs which take less time to write and modify and which are more likely to be correct.

Most data access APIs do not provide direct access to distributed data[2]. The DODS data delivery architecture describes how such a capability can be added to an API. Reimplementing an API using the DODS software client and server toolkit provides the necessary bridge between the API's function calls and the network communication infrastructure necessary to allow the API to access distributed data sets.

Once users have the power to access remotely stored data they realize that not all data are available using the API that they use. In fact, many data sets' storage formats are incompatible with third party APIs altogether because they are stored using conventions independently developed at the site where the data were first analyzed, collected, or archived. While those in charge of the data at a remote site may be very willing to provide remote access, they will probably not be willing to reformat their data for most off-site users. How do users access remote data when it is not stored in a format or data model their programs understand?

The DODS Data Access Protocol provides a means to access data sets using any one of a set of supported APIs by defining an intermediate representation for data sets, and components of data sets, that is API and format independent. Using the DODS DAP as an intermediate representation, software components can be built which translate a user program's API calls into DAP calls. Data servers can be built

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

[2] The phrase *distributed data* refers to data sets that reside on different computers which are linked by a network such as Internet. The computers may or may not be physically remote from each other. The main point is that the computers manage their data resources independently. In this paper the terms *remote* and *distributed* are used to imply independently managed resources.

which accept DAP calls and, either by translating into a third-party API or using data set-local access methods, read the data. Thus software which uses any one of a set of APIs can be used to read data so long as a DODS server for that data exists—even if the data is stored in an API different from the one used by the display or analysis software.

Note

> In a previous design, the DODS data delivery system contained reimplementations of the supported APIs which used the API entry points as the remote access interface definition; the set of calls to which a remote data server responds. However, this design did not facilitate multi API access (i.e., API translation) to data sets.

2 Rationale

Most data access APIs are targeted at application writers—they provide powerful tools to read, write, and process one specific type of scientific data and are typically specialized so that software in one application domain is easy to write. In order to fulfill this function, APIs are typically 'feature rich' (i.e., they provide many ways to perform similar tasks). This characteristic causes APIs that maintain this level of flexibility over a wide range of data types to become very complex. Because simplicity and ease-of-use are two characteristics that make APIs attractive to application programmers, most API designers try to limit the scope of an API so that it can provide the features application programmers want, at the level of detail they want, without becoming excessively large and thus burdensome to use. The netCDF and JGOFS APIs are good examples of APIs designed for a specific type of application. Each is used to access earth-science data sets, but netCDF is most suited to gridded model data, and JGOFS has direct support only for relational data.

However, the netCDF API cannot be directly used to access data sets produced by the JGOFS API and vice versa. An application using either one of these APIs would need to be modified if a user wanted to integrate both netCDF and JGOFS compliant data sets. Modifying a program so that the data access API it uses matches with the API used to write a data set is an expensive process and one which is exacerbated by the fact that more and more users do not write their own software. Modifying someone else's software can be very complex and costly at best; at worst it is not possible at all. Moreover, when the data sets are remote it may be impossible to convince their maintainers to replicate them for another API since this, too, is very costly.

One alternative to rewriting application software or replicating data is to define an intermediate representation which can be used to access data stored in any one of a number of APIs. If the intermediate representation is based on a data model that encompasses the abstractions of a certain set of APIs, then the intermediate representation can be used to translate between those APIs. The implementation of an intermediate representation interface need not be an application interface itself; instead it can be a set of tools used to build application-level components. These components can then be combined at link-time and/or runtime with user-programs to access data.

An interface which is to be used as a translation mechanism cannot afford to present the rich set of access functions that the individual APIs, which it subsumes, present. It must control the size and complexity of its interface by removing features while still providing the capability to transparently interoperate between several data models. The DODS DAP is such an interface. It is designed to be used in the construction of system components which are configured at runtime by users, maintainers or other software components. Thus, the DODS DAP is not a separate addition to the set of interfaces formed by JGOFS, netCDF, and others. Instead it is an intermediate representation which can be used to translate between those different APIs.

3 Requirements

This section presents the requirements that must be met for an intermediary representation for any of the DODS supported APIs. Currently, two APIs are supported by DODS: netCDF and JGOFS. As additional APIs are incorporated into the system (e.g., HDF), the DAP may add support for the data types those handle.

1. The DODS data access protocol(DAP) can be used to map data stored using one API to calls in a second API so long as both APIs are supported by DODS.

2. It must be possible to add new data types and operators to the DAP without changing the external interface it presents.

3. The DAP does not characterize the data set based on its native implementation. The DAP supports access to data sets that consist of a single file, several files, a relational database, a set of relational databases, or any other reasonable representation.

4. The DAP makes all data sets appear cohesive—they always appear as a single store of data regardless of their actual implementation. Not yet satisfied (23 August 1996).

5. The DAP supports data sets which consist of one or more variables.

6. When a data set contains two or more variables, the DAP makes their relationship explicit. The relations are precisely described using a syntax which other computer systems can analyze.

7. The DAP describes each variable in a data set by its:

    (a) Name

(b) Type

(c) Shape

(d) Units. Not yet satisfied (23 August 1996)

(e) Type-dependent characteristics

(f) ...zero or more attribute-value pairs

8. The characteristics of the different variable types are given by the data model.

9. The DAP can perform the following operations on a data set:

   (a) Get data set structure description

   (b) Get data set attribute list

   (c) Send arguments to a data set

   (d) Get values from the data set

4 Design

The DODS DAP design contains three important parts: A data model which describes data types that can be supported by the protocol and how they are handled, the data set description and data set attribute structures which describes the structure of data sets and the data they contain, and a small set of messages that are used to access data. Each of these components are described in the following subsections along with a summary of the runtime process configuration needed for DODS to translate from a data set API to a user program API. The discussion of the runtime process configuration frames each of the three design components and is used to explain why they are needed.

4.1 Process and Module Configuration

The set of processes and modules needed to access a data set is shown in Figure 1. In the figure two processes are shown. Process 1 contains the user program, a surrogate library implementation of the data access API used by that program, and translator component which uses the DODS DAP to request data used to satisfy calls from the user-program API calls. This process, called the *user process* communicates with the second process, the *translating server* process. The translating server contains two modules—one to recast the incoming DODS DAP calls into calls in the the data set's API and a second module that implements that API. In the figure all data is transmitted over the network using the DODS DAP regardless of the API used by the user program or used to store the data, even if the APIs are the same.

The DODS DAP's main function is to facilitate translation between two different data access APIs. In order to read data stored in API X as in Figure 1 a server which

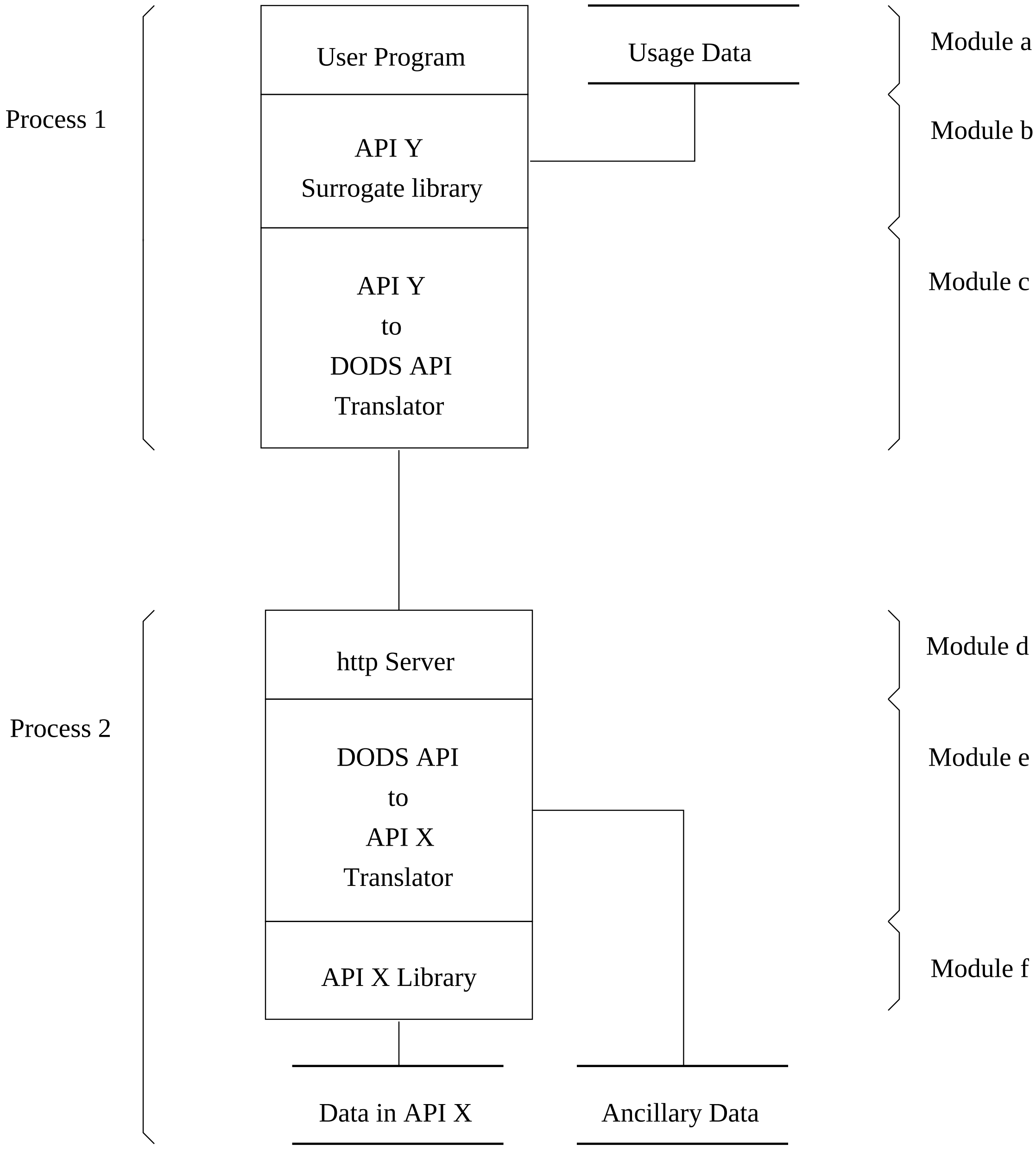

Figure 1: Two processes are used to translate API Y calls from the user program to API X calls for the data set. These same two processes are the client and server in a distributed data system. Note that DODS does not supply either the complete client or the complete server—rather DODS software consists of modules that can be combined with existing software (user programs, third party libraries) to build distributed systems. In this figure, *Module* refers to a conceptual unit (e.g., a library), not just a compile-time unit.

provides access using the DAP can use the native implementation of API X to read information from the data set.

In addition to the data accessible via the native API, each data set will contain ancillary data. These data will be directly accessed by some of the DAP calls. It will be used by both the local implementation of the DAP to aid in translating the DAP calls into the data set's native API calls and by the remote translation process.

In order to effectively translate the user program API calls into DODS DAP calls, the translator module must have some knowledge of the source data set's structure. This structural information will comprise part of the ancillary data that is accessed directly by the DODS DAP. Based on this information, which can be accessed using DAP calls, the translator can choose how to best translate the user programs data-access calls. In effect, the translator must map one data access API onto a data model to which it may not be well suited (either because the implementors of the user program or the data set have chosen an inappropriate interface).

Finally, in order to be useful by many user programs, particularly those written by a third party development team for a group of users, the translator must have some additional information about the representation of certain data objects expected by the user program. This information, called *Usage Data* in Figure 1, will allow the user to specify the format of dates and similar information which has many different common forms but no universally accepted format. It is stored in a file created by the user or developers of the *user program*. This information is defined by DODS outside of the definition of API Y and will be used by the translator module when data requests are made by the user program or when data is returned to the user program in response to one of those requests.

4.2 The Data Model

Data models provide a way to organize scientific data sets so that useful relationships between individual datum are evident. Many data models have been specifically designed to make using the data in a computer program simpler. Examples of computationally oriented data models for scientific data are hierarchical, sequential, and gridded data models.

Data models are abstract, however, and to be used by a computer program they must first be implemented by a programmer. Often this implementation takes the form of an API—a library of functions which can read and write data using a data model or models as guidance. Thus every data access API can be viewed as implementing some data model, or in some cases several data models.

Because DODS needs to support several very different data models, it is important to design it around a core set of concepts that can be applied equally well to each of those data models. If that can be done, then translation between data represented in those different models may be possible.

Currently DODS supports two very different data access APIs: netCDF and JGOFS. The netCDF API is designed for access to gridded data, but has some limited capabilities to access sequence data (although not with all of its supported programming language interfaces). The JGOFS API provides access to relational or sequence data[3]. Both APIs support access in several programming languages (at least C and Fortran) and both provide extensive support for limiting the amount of data retrieved. For example a program accessing a gridded data set using netCDF can extract a subsampled portion or *hyperslab* of that data. Likewise, the JGOFS API provides a powerful set of operators which can be used to specify which type of sequence elements to extract (e.g., only those corresponding to data captured between 12:01am and 11:59am) as well as masking certain parameters from the returned elements so that only those parameters needed by the program are returned.

The DODS DAP uses the concepts of variables and operators as the base for the data model. Within the data model, a data set consists of one or more variables where each variable is described formally by a number of attributes. Variables associate names with each component of a data set, and those names are used to refer to the components of the data set. In addition to their different attributes, it is possible to operate on individual variables or named collections of variables. The principal operation is *access*, although this can be modified in a number ways.

4.2.1 Base-Type Variables

Variables in the DODS DAP have two forms. They are either base types or type constructors. Base type variables are similar to predefined variables in procedural programming languages like C or Fortran (e.g., `int` or `integer*4`). While these certainly have an internal structure, it is not possible to access parts of that structure using the DAP. Base type variables in the DAP have three predefined attributes (or characteristics): Name, Type, and Unit. They are defined as follows:

Name A unique identifier that can be used to reference the part of the data set associated with the variable.

Type The data type contained by the variable. This can be one of `byte`, `int32`, `float64`, `string`, and `URL`. Where:

- **byte** is the same as unsigned char in ANSI C.
- **int32** is a 32 bit integer—it is synonymous with long in ANSI C when that type is implemented as 32 bits.
- **float64** is the IEEE 64 bit floating point data type.
- **string** is a sequence of bytes terminated by a null character.
- **URL** is a string as defined in DODS—Uniform Resource Locator[4].

[3] In the remainder of this document, the phrase *sequence data*, or just *sequence*, will mean an ordered set of elements each of which contains one or more sub elements where all of the sub-elements of an element are explicitly related to each other.

[4] http://www.unidata.ucar.edu/packages/dods/archive/design/urls/urls.html

Unit This contains the name of the units of the value contained in the variable. Examples of typical units are degrees Celsius, degrees Kelvin, ... If a variable is unit less, then this is null.

4.2.2 Type Constructor Variables

Type constructor variables describe the grouping of one or more variables within a data set. These classes are used to describe different types of relations between the variables that comprise the data set. This information can be useful to people who would like to understand more about the data set than can be conveyed with implicit relations. It is also designed to be useful to other programs/processes in the data access chain. There are six classes of type constructor variables defined by the DAP: lists, arrays, structures, sequences, functions, and grids. The type constructor classes besides structure provide information that is used in the translation of subsetting operations (hyperslabbing or selections and projections in netCDF or JGOFS parlance, respectively). The types are defined as:

List The **List** type constructor is used to hold lists of 0 or more items of one type. Lists of `int32`, ..., `grid` are specified using the keyword `list` before the variable's class. Access to an element of a list is possible using one of the five operators given in Table 1.

Array An **Array** is a one dimensional indexed data structure as defined by ANSI C. Multidimensional arrays are defined as arrays of arrays. In addition to element access using subscripts enclosed in brackets (`[]`), an array may be accessed using only its name to return the entire array or using a hyperslab operator to return a rectangular section of the array. In the later case, the hyperslab is defined for each dimension by a starting index, and ending index, and a stride value. Specifying a stride > 1 will cause the dimension to be subsampled by the stride value. Table 1 shows the syntax for array accesses including hyperslabs.

In addition to its magnitude, every dimension of an array may also have a name. It is possible to find the name for any given dimension (e.g., the i^{th} dimension) and thus write software which access the 3^{rd} element of the dimension *cast* (See DODS—Client and Server Toolkit[5]).

Structures A structure is a class that conveys no relational information and may contain several variables of different classes. It is used to supply information to other parts of the data access and translation system that may be useful in optimizing the access or translation operations. The structure type can also be used to group a set of unrelated variables together into a single data set.

Sequences A sequence is an ordered set of N variables which has several instantiations (or values). Variables in a sequence may be of differing classes. Each

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

instance of a sequence is one instantiation of the variables. Thus a sequence can be represented as:

$$\begin{matrix} s_{00} & \cdots & s_{0n} \\ \vdots & \ddots & \vdots \\ s_{i0} & \cdots & s_{in} \end{matrix}$$

Every instance of sequence S has the same number, order, and class of variables. A sequence implies that each of the variables is related to each other in some logical way. A sequence is different from a structure because its constituent variables have several instances while a structure's variables have only one instance (or value). Because a sequence has several values for each of its variables it has an implied *state*, in addition to those values.

Functions Functions are a subclass of Sequences and are used to indicate that one set of variables has a functional relation to a second set of variables. Variables in a function may be of differing classes. The mathematical description of this functional relation is not specified. Instead the function type is used to indicate that one of the two sets constitute the independent variables and the other the dependent variables. Typically, the variables defined by a function have more than one instance—functions are similar to sequences but have additional information about the functional dependency of variables.

Grid A grid is an association of an N dimensional array with N named vectors, each of which has the same number of elements as the corresponding dimension of the array. Each vector is used to map indices of one of the array's dimensions to a set of values which are normally non-integral (e.g., floating point values). The N (map) vectors may be members of different classes. Grids are similar to arrays of base type variables, but add named dimensions and maps for each of those dimensions.

4.2.3 Operators

The principal operation performed on any variable is to access that variable and retrieve its value or values. For a base type variable, access is the implied operation and is achieved by passing the name of the variable to a data server using the access protocol (see Section 4.6). For an instance of a type constructor class, the variable name will access the entire object and return it in an array (or C struct)[6]. Fields of type constructors may also be accessed using the dot (`.`) operator or the virtual file

[6]If a variable contains elements of different types then an instance of it *must* be returned in a `struct` and not an array. This implies that the receiving program must dynamically allocate storage for the variable and then correctly access the fields. This implies sophisticated programming (beyond the 'user' level) or an interpreter—or both.

Table 1: Classes and operators in the DAP.

CLASS	OPERATIONS
Base Type	
byte, int32, float64	`< > = != <= >=`
string	`= !=` $=\sim$
URL	`*`
Constructor	
array	`[start:stop] [start:stride:stop]`
list	`length, car, cdr, nth, member`
structure	.
sequence	.
function	.
grid	.

system syntax. If a structure `s` has two fields `time` and `temperature`, then those fields may be accessed using `s.time` and `s.temperature` or as `s/time` and `s/temperature`.

All of the classes listed in section 4.2.1 are local to the data set except the `URL` type. When an object of type URL is accessed, the server must open the data set referenced by that URL and read its values such that the structure of that referenced data set replaces the URL in the current data set. In practice this is most useful in a constraint expression when the value of a variable is compared witht he value of another variable in a different data set. If the URL cannot be referenced, then the type of the URL degrades to *string* and an access to that element of the data set returns that string.

Access to variables can be modified using selection operators. Each type of variable has its own set of selection and projection operators which can be used to modify the result of accessing a variable of that type. Table 1 summarizes the types and the operators applicable to them. In the table, operators have the meaning defined by ANSI C except as follows: the array hyperslab operators are as defined by netCDF[2] (where `a` is the start index, `b` is the stride, and `c` is the end index), the string operators are as defined by AWK[1], and the list operators are as defined by Common Lisp[3].

The hierarchy of the type constructor classes in the DAP is shown in Figure 2.

4.3 The External Representation of Variables

Each of the base-type and type constructor variables has an external representation defined by the data access protocol. This representation is used when an object of the given type is transferred from one computer to another. Defining a single external representation simplifies the translation of variables from one computer to another when those computers use different internal representations for those variable

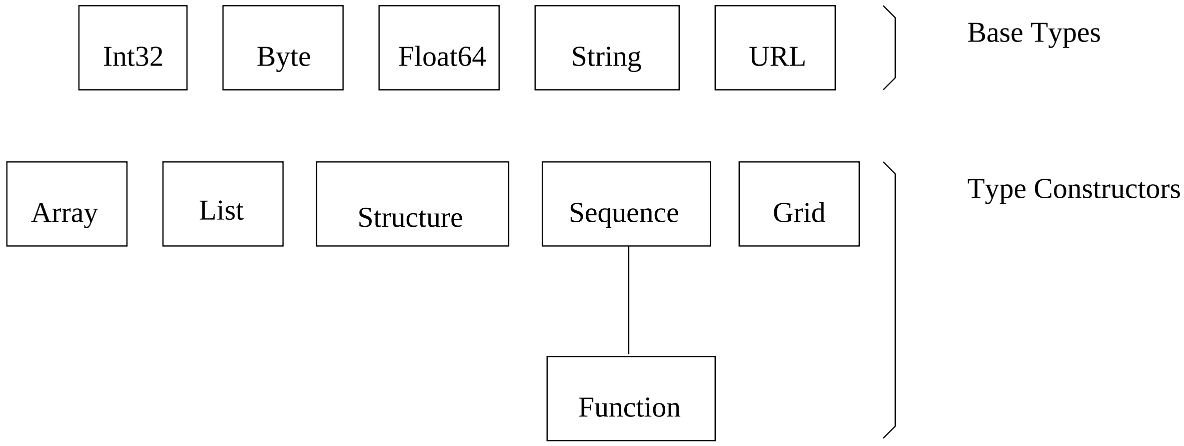

Figure 2: The hierarchy of classes in the DAP.

Table 2: The XDR data types used by the data access protocolas the external representations of base-type variables

Base Type	XDR Type
`byte`	`xdr byte`
`int32`	`xdr long`
`float64`	`xdr double`
`string`	`xdr string`
`URL`	`xdr string`

types[4].

Constraint expressions, which are an importat part of the data access protocol, do not affect *how* a base-type variable is transmitted. Constraint expressions only determine if, given a particular value for a variable, that variable should be transmitted. However, for constructor type variables contraint expressions may be used to exclude portions of the variable. In these cases, the constraint expressions can be used to change the way a particular variable is transmitted. For example, if a constraint expression is used to select the first three of six fields in a structure, the last three fields of that structure are *not* transmitted by the server.

The data access protocoluses Sun Microsystems' XDR protocol[4] for the external representation of all of the base type variables. Table 2 shows the XDR types used to represent the various base type variables.

In order to transmit constructor type variables, the data access protocoldefines how the various base type variables, which comprise the constructor type variable, are transmitted. Any constructor type variable may be subject to a constraint expression which changes the amount of data transmitted for the variable (see Section 4.6). For each of the six constructor types these definitions are:

Array An array is sent using the `xdr_array` function. This means that an array of 100 `Int32`s is sent as a single block of 100 `xdr longs`, not 100 separate `xdr longs`.

List A list is sent as if it were an array.

Structure A structure is sent by encoding each field in the order those fields are declared in the DDS and transmitting the resulting block of bytes.

Sequence A Sequence is transmitted by encoding each instance as for a structure and sending one after the other, in the order of their occurrence in the data set. The entire sequence is sent, subject to the constraint expression. In other words, if no constraint expression is supplied then the entire sequence is sent. However, if a constraint expression is given all the records in the sequence that satisfy the expression are sent[7].

Function A function is encoded as if it is a Sequence (one component after the other, in the order of their declaration).

Grid A grid is encoded as if it is a Structure (one component after the other, in the order of their declaration).

4.4 Dataset Descriptor Structure

In order to translate from the user program's API to the data set's API, the translator process must have some knowledge about the types of the variables, and their semantics, that comprise the data set. It must also know something about the relations of those variables—even those relations which are only implicit in the data set's own API. This knowledge about the data set's structure is contained in a text description of the data set called the *Dataset Description Structure*.

The data set description structure (DDS) does not describe how the information in the data set is physically stored, nor does it describe how the data set's API is used to access that data. Those pieces of information are contained in the data set's API and in the translating server, respectively. The translating server uses the DDS to describe the structure of a particular data set to a translator—the DDS contains knowledge about the data set variables and the interrelations of those variables. In addition, the DDS can be used to satisfy some of the DODS supported APIs data set description calls. For example, netCDF has a function which returns the names of all the variables in a netCDF data file. The DDS can be used to get that information.

The DDS is a textual description of the variables and their classes that comprise the entire data set. The data set descriptor syntax is based on the variable declaration/definition syntax of C and C++. A variable that is a member of one of the base

[7] The client process can limit the information received by either using a constraint expression or prematurely closing the I/O stream. In the later case the server will exit without sending the entire sequence.

Table 3: Dataset Descriptor Structure Syntax

data sets	*data set*
	data sets data set
data set	`dataset` { *declarations* } *name* ;
declarations	*declaration*
	declarations declaration
	δ
declaration	*list declaration*
	base-type var ;
	structure { *declarations* } *var* ;
	sequence { *declarations* } *var* ;
	function { `independent` : *declarations dependent* : *declarations* } *var* ;
	grid { `array` : *declaration* `maps` : *declarations* } *var* ;
list	`list`
structure	`structure`
sequence	`sequence`
function	`function`
grid	`grid`
base-type	`byte`
	`int32`
	`float64`
	`string`
	`url`
var	`id`
	var array-decl
array-decl	[`integer`]
	[`id = integer`]
name	`id`

type classes is declared by by writing the class name followed by the variable name. The type constructor classes are declared using C's brace notation. A grammar for the syntax is given in Table 3. Each of the keywords for the type constructor and base type classes have already been described in section 4.2.1. The `data set` keyword has the same syntactic function as `structure` but is used for the specific job of enclosing the entire data set even when it does not technically need an enclosing element (because at the outermost level it is a single element such as a structure or sequence).

An example DDS entry is shown in Figure 3. Suppose that three experimenters have each performed temperature measurements at different locations and at different times. This information could be held in a data set consisting of a sequence of the experimenter's name, the time and location of each measurement and the list of measurements themselves, and indicates that there is a relation between the experimenter,

```
data set {
    int catalog_number;
    function {
      independent:
        string experimenter;
        int time;
        structure {
            float latitude;
            float longitude;
        } location;
      dependent:
        sequence {
            float depth;
            float temperature;
        } temperature;
    } temp_measurement;
} data;
```

Figure 3: Example Dataset Descriptor Entry.

location, time and temperature called temp_measurement.

4.5 Dataset Attribute Structure

The Dataset Attribute Structure (DAS) is used to store attributes for variables in the data set. An attribute is any piece of information about a variabale that the creator wants to bind with that variable *excluding* the characteristics type, shape, and units. The characteristics type, shape and units are always defined for every variable; they are data type information about the variable. Attributes, on the other hand, are intended to store extra information about the data such as a paragraph describing how it was collected or processed[8]. In principle attributes are not processed by software other than to be displayed. However, many systems rely on attributes to store extra information that is necessary to perform certain manipulations on data. In effect, attributes are used to store information that is used 'by convention' rather than 'by design'. DODS can effectively support these conventions by passing the attributes from data set to user program via the DAS. Of course, DODS cannot enforce conventions in data sets where they were not followed in the first place.

The syntax for attributes is given in Table 4. Every attribute of a variable is a triple: attribute name, type and value. Note that the attributes specified using

[8] To define attributes for the entire data set, create an entry for a variable with the same name as the data set.

Table 4: Dataset Attribute Structure Syntax

attributes	*attribute*
	attributes attribute
attribute	`attribute` { *var-attr-list* }
var-attr-list	*var-attr*
	var-attr-list var-attr
	δ
var-attr	*var-name* { *attr-list* }
attr-list	*attr-pair*
	attr-list attr-pair
	δ
attr-pair	*attr-type attr-name attr-val* `;`
var-name	`identifier`
attr-name	`identifier`
attr-val-vec	*attr-val*
	attr-val-vec `,` *attr-val*
attr-val	`value`
	`identifier`
	`string`
attr-type	`Byte`
	`Int32`
	`Float64`
	`String`
	`Url`

the DAS are different from the information contained in the DDS. Each attribute is completely distinct from the name, type and value of its associated variable. The name of an attribute is an identifier, following the normal rules for an identifier in a programming language with the addition that the '/' character may be used. The type of an attribute may be one of: Byte, Int32, Float64, String or Url. An attribute may be scalar or vector. In the later case the values of the vector are separated by commas (,) in the textual representation of the DAS.

When the data access protocol is used to read the attributes of a variable and that variable contains other variables, only the attributes of the named variable are returned. In other words, while the DDS is a hierarchical structure, the DAS is *not*; it is similar to a flat-file database.

4.6 Data Access Protocol Entry Points

The DAP is a stateless protocol. Each of the DAP's entry points (i.e., the messages a data server will respond to) does a single isolated job and they can be issued in any order (although in many applications it will not make sense to get the values for a variable before finding out the name of the variable ...). The stateless nature of the DAP fits well within the context of the data delivery system described in DODS—Data Delivery Architecture[9]. In that paper a client-server architecture for remote access is described which relies on the HTTPD/CGI mechanism to build a data server. One implementation of that architecture is described in DODS—Data Delivery Design[10] uses three CGI modules, one of each of the three DAP entry points.

In this paper we talk about messages to the data server as if it is a stateful server like `ftpd`. However, that is merely a convient way to phrase the discussion of the DAP—in fact the data server is a stateless machine accessed by getting the value of a URL.

Each DODS data server must respond to three URLs; one for each of the objects (DAS, DDS and variable) which the server returns. These URLs are formed by appending a suffix to the base URL which references the data set. In addition, individual variables may be accessed using the constraint expression mechanism described here. The paper DODS—Uniform Resource Locator[11] conatins more information on this scheme.

Two messages are provided to access the data set descriptor structure (DDS) and the data set attribute structure (DAS). The response to these messages is text formated using the respective grammars in Tables 4 and 3. This text can then be parsed by the caller to determine the structure of the data set, types and sizes of each of its components and their attributes. These structures are derived both from information contained in the data set and for ancillary information supplied by the data set maintainers in separate text files (in the data delivery design the origin of these structures is described in detail). They provide information that is often referred to as 'metadata' and may be cached by the client system. Future accesses to the same data set can then skip the retrieval of these structures.

All variables are read from a data set individually using a single 'read' message. The message must include the name of the variable to read, and optionally, may include an expression that describes the range of values desired. No other information need be sent to the server. In response to this message the value(s) of the variable(s) is/are then sent back to the client. Base type variables return only a single value when accessed, but other types may return more values. For example, a `structure` with three `int32` and two `float64` members will return five values (unless an expression constrains the access).

One important capability of the DODS API, which it inherits from JGOFS, is the ability to set constraints on variables (JGOFS calls this 'using selection and projection' operators). Constraints are used to control the values and/or members of constructor types that are returned when a variable is accessed. For some data sets and some variables, constraints make little or no difference in how the variables are accessed.

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

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

[11]http://www.unidata.ucar.edu/packages/dods/archive/design/urls/urls.html

Table 5: Constraint Expression Syntax

expression	*projection selection*
projection	*variable*
	variable , *projection*
	δ
selection	*variable operator value*
	variable operator variable
	δ
compound-sel	*selction boolean-op compound-sel*
	! *selection compound-sel*
	selection
boolean-op	&

However, for certain types of data, constraining access can vastly reduce the amount of data the application needs to process and, in DODS case, transmit over the network.

Constraint expressions provide more flexibility in the way data is accessed. In a data set with many interrelated variables, or with very large variables, these expressions are a way for the user program to move some of the complex logic needed to search data, in order to find those data with some desired set of properties, away from the user program and into the data supply system. Different types of constraint expressions make sense for different types of data. The grammar for constraint expressions is given in Table 5. In the table *Operator* is one of the variable-class dependent operators listed in Table 1, ! is boolean *not* and & is boolean *and*.

5 Conclusion

The Data Access Protocol is used to access data sets within the DODS data delivery system. It is not intended to be limited to DODS, but it is specialized for the system. Among those specializations are the stateless nature of the protocol. While the DAP could be used in many different contexts, it is principally designed to be the transmission protocol used by the data servers in the DODS data delivery system.

DODS supports access to data sets stored in a number of established, third party APIs. In addition, it supports cross-access to those data sets. Thus data in any one of the supported APIs is accessible using any one of the supported APIs (at least in principle—some *possible* accesses may wind up returning meaningless, or null, data). The DAP facilitates this cross connections of user programs and data sets by providing a common access protocol for each API. Each data server will provide access to data using the DAP and each surrogate library (See DODS—Data Delivery Architecture[12]

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

and DODS—Data Delivery Design[13]) will satisfy the supported API's function calls with information obtained via the DAP.

To support its stateless interface, the DAP consists of two structures which contain attribute and type information about the variables in the data set and a single, programmable, mechanism for reading a variable. The attribute and type information structures can be used to build powerful translators which read variables in one type and transform them, on-the-fly, to types suitable for the surrogate API.

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

References

[1] Brian W. Kernighan and Rob Pike. *The Unix Programming Environment.* Prentice-Hall, Inc., Englewood Cliffs, New Jersy, 1984.

[2] Russ Rew, Glenn Davis, and Steve Emmerson. *NetCDF User's Guide.* Unidata Program Center, Boulder, Colorado, April 1993. Version 2.3.

[3] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, Bedford, Massachusetts, 1984.

[4] Sun Microsystems, Mountain View, California. *XDR.* Version 4.