DODS Programmer’s Guide
The Client and Server Toolkit
Version 1.2

James Gallagher
Tom Sgouros

July 1, 2004
This document describes how to use the OPeNDAP toolkit software to build OPeNDAP data servers, clients and client-libraries. Using the objects and functions contained in the toolkit, you can create programs which serve data over the internet as well as programs that can request data from any OPeNDAP server.

This document covers release 3.2 and later of the DODS software.


0.1 Who is this Guide for?

This guide is for people who wish to use the OPeNDAP software to write a new OPeNDAP data server, a new client, or a new client library. Typically, this will only be those people who wish to serve data in a format that is not currently supported by the DODS team, or who have an existing application that uses an idiosyncratic or unusual API for data access. Most people will be able to use one of the already written servers or client libraries. See the *The OPeNDAP User Guide* for a list of these.

This documentation assumes that the readers are C++ programmers, are familiar with networked applications, and the POSIX programming environment. The DODS/OPeNDAP project also provides a native Java class library (API) that parallels the C++ software described here.\(^1\)

Also available are two tutorials, *Writing an OPeNDAP Client* and *Writing an OPeNDAP Server*, which describe how to write a client or a server, respectively.

Because the type of information presented in a document like this depends to a large extent on the needs of its readers we welcome your feedback and comments. In particular, if you have any questions about individual sections, email those questions and we’ll send back an answer as well as including that information in the next version of this document. Send queries to: support@unidata.ucar.edu.

\(^1\)While this manual describes the C++ toolkit in detail, all of the concepts and much of the structure can be directly translated to the Java toolkit.
0.2 Organization of this Document

This Guide is divided into five chapters.

Chapter 1 provides background information on the organization of the toolkit software.

Chapter 2 describes how to use the Network I/O classes to manage virtual connections.

Chapter 3 discusses how to sub-class the toolkit C++ classes so that they are specialized for your specific use.

Chapter 4 describes in detail how to write certain sections of both the data server and the client-library for a new API.

Chapter 5 describes how to link user programs with the new client-library implementation of an API.

Chapter 6 contains complete descriptions of all the DODS classes.

0.3 Conventions

The typographic conventions shown in Table 1 are followed in this guide and all the other DODS documentation.

<table>
<thead>
<tr>
<th>Literal text</th>
<th>Typed by the computer, or in a code listing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>User input</td>
<td>Type this precisely as written.</td>
</tr>
<tr>
<td>Variables</td>
<td>Used as a place holder for a user-specified or variable value. Choose an appropriate value and use that in place.</td>
</tr>
<tr>
<td>Button Text</td>
<td>Used to indicate text on a GUI button.</td>
</tr>
<tr>
<td>Menu Name</td>
<td>This is the name of a GUI menu.</td>
</tr>
</tbody>
</table>

When referring to a button in a menu, we will often use the notation: [Menu,Button]. For example, [Options,Colors,Foreground] would indicate the [Foreground] button in the Colors menu, selected under the Options menu.
4.1.4 The Usage Filter .................................................. 38
4.1.5 Documenting Your Work ......................................... 39
4.2 Client Libraries ....................................................... 43
  4.2.1 Rewriting the Open and Close Functions ...................... 43
  4.2.2 Getting Information about Variables .......................... 44
  4.2.3 Reading the Values of Variables from a Dataset ............. 44
  4.2.4 Functions that Write to Data Sets ........................... 46
  4.2.5 Adding Local Access to a DODS Client Library ............... 46
4.3 Using Constraints .................................................... 47
  4.3.1 How Constraint Expressions are Evaluated .................... 47
  4.3.2 Different Ways of Using Constraint Expressions .............. 47
5 Linking Your Program ................................................ 51
6 DAP C++ Reference .................................................... 53
A Overview of the DODS Server Architecture .......................... 55
  A.1 Outputs ............................................................ 56
    A.1.1 HTML Data ................................................. 56
    A.1.2 ASCII Data (Text) ........................................ 57
    A.1.3 Binary Data ............................................... 57
  A.2 Inputs ............................................................ 61
    A.2.1 Request types ............................................. 61
    A.2.2 Constraint expressions ................................. 62
    A.2.3 Server functions ....................................... 62
B Overview of the DODS Client .......................................... 67
List of Figures

1.1 A DAS Object ................................................. 11
1.2 An Attribute Table ......................................... 12
1.3 A Nested Attribute Table ................................. 13
1.4 The DDS Object ............................................ 14
1.5 A sample Grid ............................................... 15
2.1 The Connect Class ........................................ 20
3.1 The subclass of Connect used with the NetCDF client library. 27
3.2 The recoded open call of NetCDF ......................... 28
4.1 A simple DODS data server dispatch CGI ............... 33
4.2 The DAS filter program .................................... 35
A.1 The DODS Data Document and the DDS ............... 63
A.2 Parts of a DODS URL (without a constraint expression) . 64
A.3 The Architecture of a DODS Data Server ................ 65
B.1 The Original Program ...................................... 68
B.2 The Modified Program .................................... 69
B.3 Another Way ................................................. 70

List of Tables

1 Typographic Conventions ................................. v
1.1 Table of relational data .................................. 10
A.1 Table of URL suffixes .................................. 61
The Distributed Oceanographic Data System (DODS) is a system used to facilitate access to scientific data on the internet. Using DODS, you can turn any data analysis program that uses one of several data access APIs into a powerful, internet-friendly data browser. For basic information about the structure of DODS and its components, please see The OPeNDAP User Guide.

The DODS Toolkit software consists of a collection of C++ classes used to build DODS data servers and clients:

**Type classes** A complete implementation of the DODS data access protocol (DAP). This consists of a set of virtual classes for different types of data, which must be sub-classed to use.

**Data classes** This is a set of classes, including both the Data Descriptor Structure (DDS) and the Data Attribute Structure (DAS), designed to contain information about a dataset’s data.

**Connect classes** These classes are used by a DODS client to mimic a durable connection to a DODS server.

**DODSFilter** This class consists of several utility functions useful for writing DODS servers.

**CGI Utilities** The file cgi_util.h describes a small number of other common functions needed by DODS servers.

The DAP toolkit contains three class hierarchies: one for each of the DAS and DDS objects, rooted in classes with those names, and one for the variables, rooted at the class BaseType. The following sections present an overview of these class hierarchies and describe, in general terms, how they are used to build the client-library and data server. See Section 1.1 on page 3 for information about the
DDS and DAS classes. To use the type classes, they must be subclassed. Chapter 3 provides a detailed description of this process.

Detailed information about the DAS and DDS themselves is available in the *The OPeNDAP User Guide*. Descriptions of the classes and their member functions are in the reference material, Chapter 6.

The DODS toolkit also contains two classes used by DODS clients to manage network connections between themselves and a server. The `Connect` object manages a single connection with some DODS server, and the `Connections` object manages a group of `Connect` objects. The `Connect` class is meant to be subclassed when used by a real client library. See Chapter 2.
1.1 The DAS and DDS Objects

The dataset attribute structure (DAS: page 4) and dataset descriptor structure (DDS: page 6) objects are used to store information about a data set’s variables. These objects are used on both the client and server sides, although there are class features that only pertain to one or another of the roles. They can be thought of as metadata objects. In this book, however, we will avoid the term metadata because often this is data to many users.

It might be said that neither the DAS nor the DDS contain actual science data — the DAS contains attribute information from the data set while the DDS contains structural information about the data set and variables in the data set. Since the boundary between data and metadata (or data attributes) is often a blurry one, this is not a distinction we will insist on.

To build both the DAS and DDS, the server either reads information directly from the dataset or from DODS-specific ancillary data files, depending on the capabilities of the data access API used to access the data. The DAS and DDS server filter programs do this and then transmit the resulting object to the client.

On the client side, the DODS client uses information in the DAS and DDS to satisfy API calls issued by the user program requesting information about variables, their type, shape, and attributes. The client requests both of these objects when it first contacts the remote data set. The DAS and DDS objects are then stored as part of a virtual connection to that data set and can be used repeatedly by the client library without retransmission.

The DAS and DDS objects have both an internal and an external representation. Internal to the DODS client or server, these structures are stored as C++ objects, while their external representation is as text. The object is sent from the server to the client using this text representation. Each of the two classes contains a parser which can read the text representation and recreate the object’s internal representation. In addition, it is possible to write the text representation for either object (using a text editor) and then use the parser to create the internal, C++, object. Furthermore, the text representation is a type of persistence and can be used to build a flexible object caching mechanism.

One possible use for this caching mechanism is to store the DAS and DDS for a dataset and use the stored versions in place of opening the data set and reading information about it and its contents. For large data sets with many variables this can result in a significant performance improvement, and several of the packaged DODS servers use it.

The caching mechanism may also be used on the server-side to store extra information about the data set—information that is not present in the data set.

---

1 Whatever it is. The DODS client can be another server, a user application linked with a DODS-compliant API, or a standalone program using the DODS data access protocol API. In any case, the use of the class libraries described in this document is identical.
proper, but which the data provider would like included when people access the
data set via DODS. In this scenario, the data server first integrates the data set
file(s) using the API and builds the DAS and DDS. Once an initial version of the
DAS and DDS are resident in memory, the parser is used to read an external text
file which contains additions to, or corrections of, the information extracted from
the data file. This information is the ancillary data introduced above. Several of
the DODS servers use this mechanism.

1.1.1 The DAS Object

The DAS contains attribute information that is generally not used by software
when processing the variables; this object is specifically designed to hold all the
information in a data set that has no where else to go. Each variable can have an
unlimited number of attributes. There are also ‘global’ attributes that apply to the
dataset as a whole. Each attribute is a set of three elements: the attribute name,
type and value. The supported types for an attribute are: Byte, Int32, Float64,
String and URL, and vectors of these types. These types have the same range of
values as the corresponding types in the data access protocol.

A variable’s attributes can be any qualities that are not part of the DDS. Thus, the
type and shape of a variable are not attributes; they are characteristics of the
variable and are part of the DDS (see Section 1.1.2). However, users may store
any other information as attributes\(^2\), including paragraphs of text, vectors of
integers and floating point values, and so on. There are some standard attributes
required by the DODS standard. The system could work without these attributes,
but they are required anyway to make other aspects of the systems work better.

Many APIs support the concept of information in the data set that is not
structured, and provide function calls to access that information. The DAS object
provides a place for all that information. The data server can interrogate the data
set and build the DAS and the client-library can use the DAS object to satisfy many
of the API calls requesting information about the variables in a data set.

Figure 1.1 is a diagram of the DAS object. The DAS consists of the following
components:

► A list of attribute tables, each of which is a list of attributes for a particular
data variable. Since data variables can contain other data variables (as with
a compound data type, for example), an attribute table can contain other
attribute tables.

► A parser and scanner to read a text version of the DAS object and create the
corresponding C++ (binary) representation in memory.

► A printer with which you can create a text version of the DAS object.

\(^2\) Actually, there is no reason that type, etc. cannot be stored as an attribute; however, it must be
in the DDS regardless
1.1 The DAS and DDS Objects

The DAS printer (DAS: :print()) is used to create a textual representation of the C++ object. The object is transmitted from server to client using this representation. The scanner and parser (DAS: :parse()) reads the printed representation and creates a C++ object to match this specification. This object method can be used to read the text version from a disk or from a network connection, depending on the input stream identified for it.

The third part of the DAS, the attribute table list, points to a series of attribute tables. An attribute table is a list of name-type-value triples used to describe a data variable. Each attribute describes some aspect of the data variable. The example list in figure 1.2 shows what a table might look like for one of the data variables in figure 1.1. (The types have been left out of the diagrams for clarity.)

The DAS object contains member functions to add or retrieve AttrTable objects as well as individual attributes based on a variable name. In addition, the AttrTable object may be traversed using a Pix.\(^3\)

The value of an attribute can itself be another attribute table. So, for example, an aggregate variable that contains other variables, such as a Structure, might have an attribute table for each of its member variables. In figure 1.3, you can see illustrated the attributes of an aggregate data variable. The first three attributes apply to the collection of data (it was taken with a CTD instrument, from the good ship R/V Endeavour, and so on), while the next three attributes reveal attributes of the constituent data variables. Each of the values of these attributes is itself an attribute table, containing attribute data about that data type.

DODS attribute tables are modelled with the AttrTable class. Each AttrTable object contains a doubly-linked list of attribute triples, of a structure named entry. Each entry object contains a Name, Type and vector of values. AttrTable provides methods for reading, writing, and modifying the attribute table.

The DODS definition of dataset attributes contains a “Global” attribute. This has nothing to do with the data structure of the DAS object, but with its use in DODS. Global attributes apply to all the variables in the dataset. They can also be thought of as being attributes of the dataset itself.

\(^3\)A Pix is a “pseudoindex” object. See the libstdc++ documentation for more information.
1.1.2 The DDS Object

The DDS is used to store information about the organization of the data set and its variables. It contains information about the type and shape of variables. While the DDS is similar to the DAS in that it is used to store information about the data set, it is used quite differently by both the client and server components of DODS. The DAS is a stand-alone object and is used solely for the purpose of storing attributes of variables and the dataset. The DDS, however, stores type information about a data set’s variables by storing actual instances of those variables.

The DODS data access protocol variable objects have methods that can be used to read values from a data set or transfer the variable’s value over the network. This makes it convenient to use the DDS object itself to hold data, and on the server side, the DDS object is used by both the DDS filter program and the data filter program. (See Section 4.1 on page 31 for more information about the structure of the DODS server.)

Figure 1.4 shows the structure of the DDS class objects. The DDS consists of the following components:

- A String object used to store the name of the dataset to which this DDS refers.
- A singly-linked list to store the variables in the data set. Each variable is stored in an instance of one of the BaseType class’s descendants, and can be a simple or compound data type.
- A “printer” method to create a textual representation of the DDS. Among other uses, this is used to transmit the DDS from server to client.
- A parser and scanner to read a text DDS and convert it into its C++ form. This is used to read DDS information from a disk, and also to receive it over the network.
- A singly-linked list of parsed constraint expression clauses. The clauses wait here to be evaluated.
- A constraint expression parser and scanner to extract constraint expression clauses from the input constraint expression. This component is responsible for creating the list of constraint expression clauses.

The DDS object provides methods to access and operate each of these components. The two lists can be traversed with a g++ Pix object.

The DDS is ‘lexically scoped’ so that two Structure variables may have components with the same names; each component will be referred to using the Structure name and the dot operator. So for example, if a DDS called ralph contains two structures, vitals and new_hip, and each structure contains a variable called age, you can differentiate the two by referring to one as ralph.vitals.age and the other as ralph.new_hip.age.
1.2 The Type Hierarchy

The Type Hierarchy is the set of classes that form the hierarchy used to build objects that contain data. These classes comprise the data model for DODS. They contain simple data types such as integer and floating point values as well as compound types like structure and sequence. Each type is embodied by a C++ class, and the classes are arranged in a class hierarchy, with a BaseType defining properties inherited by all the type classes.

This section contains a brief description of the different types, their relation to one another, and how they are used in an application program. For detailed descriptions of the characteristics of each type, including inheritance diagrams, please see Chapter 6.

The DODS types can be divided into four categories:

- **BaseType**
- **Simple Types**
  - Byte  Int16  UInt16  Int32  UInt32  Float32  Float64  Str  Url
- **Vector Types**
  - List  Array
- **Compound Types**
  - Structure  Sequence  Grid

Unlike the other classes in the DODS toolkit, the type classes are abstract classes—in order to be used by a program, you must subclass the hierarchy and create concrete classes to instantiate.

1.2.1 Common Ancestor: BaseType

The root of the type hierarchy is the abstract class BaseType. This class, because it is abstract, is never instantiated itself. BaseType is used as the base class for all of the different types of variables, and contains common member functions used by all the other type classes. For simple variables such as Int32, only the abstract virtual functions in BaseType need to be added to complete the class definition. Compound types like Structure usually require more. A compound type contains one or more instances of BaseType, and requires methods to access, add and remove these member variables.
1.2.2 Simple Types

The DODS simple data types consist of Byte, Int16, UInt16, Int32, UInt32, Float32, Float64, Str and Url. These data types match very closely the corresponding types in C or Fortran. Note that—internally—each of these types uses either the C or C++ representation to hold the value of the object.

These abstract classes are direct descendents of BaseType which contain only their definitions for BaseType’s abstract member functions and a single constructor.

**Byte** Variables which store bytes. Equivalent to unsigned char on most UNIX workstations.

**Int16** Variables which store integer values as 16-bit two’s-complement signed integers. Equivalent to int on most 16-bit UNIX workstations.

**UInt16** Unsigned integer. A 16-bit unsigned integer value.

**Int32** Variables which store integer values as 32-bit two’s-complement signed integers. Equivalent to int on most 32-bit UNIX workstations.

**UInt32** Unsigned integer. A 32-bit unsigned integer value.

**Float32** Variables which store floating point data. Defined as the IEEE 32-bit floating point data type, equivalent to a float in ANSI C.

**Float64** Variables which store floating point data. Defined as the IEEE 64-bit floating point data type, equivalent to a double in ANSI C.

**Str** Variables which store string information. A DODS string is not a sequence of characters referenced using a pointer (as it is in C), it is represented using a C++ object of the class String.

**Url** Variables which store references to network resources. This is a sub-class of Str.

1.2.3 Vector Types: Array, List

The vector data types are Array and List. A List is a simple ordering of elements of a single type. An Array arranges the elements so that they can be easily accessed with one or more indices.

**Array** Instances of Array have different semantics than arrays in C. They are multi-dimensional data structures which contain N instances of some data type. Each instance in the array can be referred to by an integer index between 0 and N-1. Multidimensional arrays are declared using C’s syntax of a sequence of bracketed integer values: Int32 a[10][20] declares an array of 10 arrays of 20 integers. However, unlike C arrays, the Array class
The Type Hierarchy

supports named dimensions. In the preceding example, the array could have been declared: Int32 a[row = 10][col = 20] where row and col are the names of the first and second dimension, respectively. You can use the dimension_name and dimension_size member functions of the Array class to determine the name and size for the $i^{th}$ dimension.

Array, like all of the compound types, contains a reference to a component variable. In the preceding example, the instance of Array would contain information about the dimensions of the array (10 by 20), but not the type of the elements (Int32). The element type information is stored in the component variable which the instance of Array references.

When creating an array, the dimension sizes (and optionally their names) must be set. Regardless of the shape of the array, it is always stored as a vector. In order to access the element of a multidimensional array it is necessary to calculate the offset for a given element.

List A List is an ordered collection of elements of unknown length. This is in contrast to the Array, whose size is always known in advance. When a List type is declared no size information is supplied, but in order to transmit a List object the length of that list must be known. Thus, internally the number of elements in the current value is stored.

1.2.4 Compound Types: Structure, Sequence, Function, Grid

The compound data types are used to build new types as aggregates of other types, including other compound types. (Note that List and Array are compound types, as well, but contain only a single type of data.) Structure, Sequence and Function all contain a list of BaseType objects. However, they have different semantics; a Structure is a simple aggregate; nothing other than aggregation is implied, while Sequence and Function define templates for relational objects. A Grid combines several Array objects so that nonlinear values may be applied to the indices of an array.

Structure A Structure is an ordered collection of variables that conveys no relational information other than grouping. The variables that are members of a Structure may be of different types. In addition to the (possible) benefit of added organization, Structure may be used to supply information to the system that may be useful in optimizing the access or translation operations.

Sequence A Sequence is similar to a Structure in that it consists of an ordered collection of variables which may be of different types. However, where an instance of a Structure object describes a single set of data variables, an instance of a Sequence object describes a set of data variables, each of which is an entry in an ordered series of similar data variables.

Consider a Sequence named $S$, where each instance is called $s$:
Every instance $s_i$ of $S$ has the same number, order, and class of variables. A *Sequence* implies that each of the $n$ variables is related to each other in some logical way. Because a *Sequence* has several values for each of its variables it has an implied state, or position in the sequence, in addition to the instance data values.

Table 1.1: Table of relational data.

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>James</td>
<td>32</td>
<td>165</td>
</tr>
<tr>
<td>Charlie</td>
<td>7</td>
<td>65.4</td>
</tr>
<tr>
<td>Bob</td>
<td>10</td>
<td>80</td>
</tr>
</tbody>
</table>

For example given the information in Table 1.1, $s_0$ is James, 32, 165, $s_1$ is Charlie, 7, 65.4, ... The data in the table might have the following Sequence declaration:

```plaintext
Sequence {
    Str name;
    Int32 age;
    Float64 weight;
} people;
```

**Grid** A *Grid* is an association of an $N$ dimensional *Array* with $N$ named vectors (map 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). Two map vectors may be members of different classes.

In figure 1.5, the grid element indicated by `Grid[2][3]` corresponds to $N[2]$ and $M[3]$, or $N = 92.3$ and $M = 3.9$ respectively. The element has a value of 29.7.
1.2 The Type Hierarchy
1.2 The Type Hierarchy

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>Grid</th>
<th>Grid</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.2</td>
<td>32.0</td>
<td>32.0</td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>31.5</td>
<td>31.5</td>
</tr>
<tr>
<td></td>
<td>1.8</td>
<td>31.1</td>
<td>31.1</td>
</tr>
<tr>
<td></td>
<td>3.9</td>
<td>30.8</td>
<td>30.8</td>
</tr>
<tr>
<td></td>
<td>4.5</td>
<td>29.2</td>
<td>29.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N=</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>67.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>68.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>92.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>95.2</td>
</tr>
</tbody>
</table>

Figure 1.5: A sample Grid.
Because DODS uses a communication protocol (http) that does not maintain state information about the link between two processes, these connections are virtual, and all the information about them is maintained by the client. The toolkit contains two classes to help clients manage connections to one or more DODS servers. One class, `Connect`, stores information used when a connection is established. It also allows a program to provide local access to data (without a DODS data server).

The second network class of the DODS Toolkit, `Connections`, manages instances of the `Connect` class. It provides a mechanism for the client libraries to pass back to user programs the type of object (such as `int`, opaque pointer, ...) they expect and then to use one of those objects to access the correct instance of `Connect`. `Connections` is a template class. That is, a client library uses the `Connections` class to make an array of some sub-class of `Connect`. 

2

Managing Connections
2.1 Connect

The Connect class manages one connection to either a remote data set, via a DODS data server, or a local access. For each data set or file that the user program opens, there must be exactly one instance of Connect. Because information needed for local access is stored in an instance of Connect, this class may also be used (sub-classed) for each client API that needs to maintain additional information about the connection.¹ In the API-specific child of Connect additional members can be added to store state information that the client-library needs to maintain the virtual connection (See Section 3.2 for more information on making virtual connections).

The Connect class is illustrated in figure 2.1. These objects contain the following information:

- **URL** The URL with which the client library opens the connection with the DODS server.
- **DDS** The DDS of the opened dataset. This must be retrieved from the dataset.
- **DAS** The DAS of the opened dataset. This must be retrieved from the dataset.
- **Gui** The Gui object indicates a graphical widget on the client platform that displays information about the data transfer in progress.
- **Error** The Error object contains information used to help the client process an error message from the server.

The Connect class provides member functions to get the dataset’s DAS, DDS and data. The instance of Connect (or a subclass of Connect) stores the URL as the user provides it.

When the client library receives a URL via its “open” call (which will not actually be called open but performs the function of opening a file or data set; e.g., NetCDF’s ncopen) it passes that URL to the Connect member functions like request_das and request_dds. These member functions append an appropriate extension (.das and .dda, for example) onto the URL and retrieve the resulting information from the server, the DAS or DDS for the dataset.

If you are re-implementing an API and must support function calls that modify how data is accessed (e.g., by creating array slices or by choosing one of a set of variables), then you will need to translate those requests into a DODS constraint expression. You would then pass these synthesized constraint expressions to the

¹For example, the subclasses JGConnect and NCConnect exist for JGOFS and NetCDF, respectively.
Connect::request_data member function. (See Section 4.3 on page 47 for more information about constraint expressions.)

The specialized version of Connect is the place to put state information needed by a recoded API or other client. This can be used, for example, to emulate an API that maintains a list of open files.

2.2 Connections

The class Connections is used to manage a set of instances of the class Connect by providing a means to map an index or opaque pointer to an instance of Connect.

When a new instance of Connect (or a descendent of Connect; See Section 3.2) is created, it is added to the Connections object using the add_connect member function. add_connect returns an integer that can be used to access that instance of Connect at any time. Similarly, when an instance of Connect is to be deleted, the object can be referred to via the Connections object and this index.
In order to be used by real client-libraries and data servers, many of the classes in the DODS toolkit must be subclassed. For example, the Type Hierarchy classes, which represent the data types in the DODS data model, are all abstract classes. In order to use them in a program they must be subclassed. The `Connect` class also must be subclassed if it to hold additional information about the connection.
3.1 Sub-classing the Type Hierarchy

In order to link a program with the DODS Toolkit library, the DAP's abstract classes must be subclassed and those subclasses must ensure that all of the member functions of those classes have valid definitions. This is necessary because of C++'s rules governing abstract classes. The next three sections cover sub-classing the simple, vector and compound classes, respectively. In addition, a sample set of classes (called the Test classes because they are used by the DAP tests) is included with the DAP distribution. You can read the source code for those classes to find out how they were created.

Each of the sub-classed types must supply:

- A constructor that takes a String argument and returns an object with that name.
- A virtual destructor.
- A copy function called ptr_duplicate.
- The read function, to read data from a disk and load it into the type class object.

The ptr_duplicate member function returns a pointer to a new instance of the type of object from which it was invoked. This member function exists so that objects that are referenced through pointers to BaseType can correctly copy themselves. (If you were to use the operator new to copy an object referenced through BaseType, you would get a BaseType, not a new instance of the type of the referenced object.) Note that ptr_duplicate is a virtual function so an object which is a descendent of BaseType will get the most specific definition of that function.

The read function is much more complicated to write than ptr_duplicate, and the difficulty varies depending on the data type to be read. However, this function is only used on the server side of the system and not by the client. That is, it can be implemented with a null function body if all you are building is a client library. The read function takes two arguments, the dataset name, and an error flag. It must read from that dataset the values specified by the current constraint expression. The error flag is passed by reference so that read can set its value and callers can test it. The class BaseType contains a number of member functions to facilitate writing a read function. The return value of read is TRUE is there is more data to be read (by additional calls to read) or FALSE otherwise.
3.1 Sub-classing the Type Hierarchy

3.1.1 Sub-classing the Simple Types

Creating a read function for the simple type classes is a fairly straightforward operation. Simply read the function using the data access API’s standard protocol, and use the type class’s val2buf function to load the data into the type object.

The following function is taken from the DODS implementation of the NetCDF data access library. (See the file src/nc-dods/NCByte.cc.)

```cpp
NCByte::read(const String &dataset, int &error)
{
    int varid; /* variable Id */
    nc_type datatype; /* variable data type */
    long cor[MAX_NC_DIMS]; /* corner coordinates */
    int num_dim; /* number of dim. in variable */
    long nels = -1; /* number of elements in buffer */
    int id;

    if (read_p()) // already done
        return true;

    int ncid = lncopen(dataset, NC_NOWRITE); /* netCDF id */
    if (ncid == -1) {
        cerr << "ncopen failed on " << dataset<< endl;
        return false;
    }

    varid = lncvarid( ncid, name());
    (void)lncvarinq( ncid, varid, (char *)0, &datatype,
        &num_dim, (int *)0, (int *)0);

    if(nels == -1){
        for (id = 0; id < num_dim; id++)
            cor[id] = 0;
    }

    if (datatype == NC_BYTE){
        dods_byte Dbyte;
        (void) lncvarget1 (ncid, varid, cor, &Dbyte);
        set_read_p(true);
        val2buf( &Dbyte );
        (void) lncclose(ncid);
        return true;
    }

    return false;
}
```

The following points are worth consideration about the above example.
Check to see if this variable has already been read.

The `lncopen()` function call is simply the `ncopen()` function from the NetCDF library. Also, the `lncvarid` function is renamed `ncvarid` and so on. The names have been changed to avoid link-time problems. (Remember that the whole point of this exercise is to create a new `ncopen()` function and its friends.

This sets the flag tested with the `read_p` function.

This command transfers values from the dataset's variable to the DODS instance. This is the point where the read actually happens.

Note the use of `dods_byte` in the code example. The DODS configuration process creates definitions for the simple data types like this one. They are stored in `config_dap.h`.

The `read` member function is used by the constraint expression evaluator to extract data from a dataset during evaluation of the constraint expression. This is particularly important to remember because the `read` member function for a simple data type will be called when reading an aggregate type such as `Structure`.

### 3.1.2 Sub-classing the Vector Types

The vector data types require the same abstract member functions be defined as the simple types. The definition for `ptr_duplicate` is also the same for vector as for simple types. However, the `read` member function for the vector types (classes `Array` and `List`) is more complicated than for the simple types because vectors of values are represented in two ways in DODS, depending on the type of variable in the vector. Arrays are stored as C would store them for the simple types such as `Byte`, `Int32` and `Float64`. However, compound types are stored as arrays of the DODS C++ objects (with the exception of arrays themselves, but more on that later).

When reading an array of `Byte` values, the `val2buf` member function should be passed a pointer to values stored in a contiguous piece of memory. For example, when `read` is called to read a variable `byte-array`, it must determine how much memory to allocate to hold that much information, use `new` to allocate an adequate amount of memory, use the dataset’s API calls to read `byte-array` into the newly allocated memory and then pass that memory to `val2buf`. This same procedure can be followed for all the simple types.

However, when reading an array of `Structure`, for example, the values must be stored in the DODS `Array` object one at a time using the `Array` member function `set_vec`. An `Array` object containing a 3x4 array of `Structure` objects will actually point to 12 different instances of that class. An `Array` object containing `Byte` objects contains only one instance of the `Byte` class, as a template for the array elements.
Arrays in DODS are unlike arrays in C in that an array object may have more than one dimension. In terms of the way a value is stored, however, an Array is a single dimensional object. When an Array is declared as having two or more dimensions, those are mapped onto a single vector. To access the element $A_{ij}$ of array $A$, you must know the size of the first dimension, $I$, and use the expression $i \times I + j$ to compute the offset into the vector.

Since the class List is a single dimension array without declared size (the size of each value of the List object is stored in the object) the rules for Array’s read member function apply.

### 3.1.3 Sub-classing the Compound Types

The read member function of the compound data types simply iterates over the contained variables calling their read member functions. In the future, this definition will move into the supplied classes (That is, read will no longer be a abstract member function for the compound types.).

The two constructor types Sequence and Function are different from all the other types in the DAP in that they have state. That is, the value of a sequence depends on how many values have been read previously. This is very different from an array where the $i^{th}$ element has the same value regardless of what has happened before. When you write implementations for read in the Sequence and Function classes, you must be sure to write those member functions so that they can be called repeatedly and that each call to read returns the next value of the corresponding data Sequence or Function.

This is true because the constraint expression evaluator must be able to apply certain constraints to values of individual sequence elements and is actually implemented in the DDS class by first calling the read member function, evaluating the constraint expression based on the values and, if they constraint expression is satisfied, calling the serialize member function. See the member function DDS::send (See section 4.3.1 for more information on evaluation of constraint expressions).

If, for some reason, it is not possible to write read so that it gets called once for each sequence value\(^1\), then you must re-implement DDS::send so that its functions are performed. For example, you could implement Sequence::read so that the entire sequence is read in and overload Sequence::serialize and Sequence::deserialize so that the next set of values are sent/received. You would then build a send that called the Sequence::read member function once and extracted each successive value, evaluated the constraint expression using on that value and used the result of that evaluation to determine whether to send the value or not.

---

\(^1\) A single entry in a sequence, modelled as a row in a relational table, is sometimes called an instance of the sequence. This is useful terminology, but is occasionally confusing when we are also talking about instances of objects.
3.2 Sub-classing the Connect Class

The DODS API defines a connection-less protocol in which a server keeps no information for a client in between data requests. This is in contrast to most data access APIs, which maintain state information about the files or datasets that a user program currently has open. To simulate an API’s connection, the client library for that API must create a virtual connection using information about the data set it has read from the server. That is, the client library must maintain the illusion of a connection (state) for each open data object (typically a file) even though no such connection actually exists.

DODS provides the Connect class to make this a simple process. This class contains a variety of information about a dataset and its location on the internet, including the dataset’s URL. Most of the information necessary to fake a connection is contained here.

Data access APIs differ widely however, and it is usually necessary to add some information to the Connect class to make a workable virtual connection. This information can be stored in the subclassed Connect. The API’s “open” call must be completely recoded so that the DAS and DDS objects, as well as any other information, are requested from the dataset server and stored locally in whatever form is most convenient.

Figure 3.1 shows the sub-class of Connect used to store information extracted from the DAS and DDS objects (for the NetCDF library), and used to simulate the storage of information in the original API.

The private data added to the Connect object with this class is as follows:

- `ncid` This is only used for access to local files. It is the NetCDF file descriptor, or file ID.
- `nvars` The number of variables in data file.
- `ndims` The number of dimensions found. Note that not all the variables in a NetCDF file have the same number or set of dimensions. This number is the list of all the different ones.
- `dim_name[MAX_NC_DIMS]` The names of the dimensions found in the data file.
- `dim_size[MAX_NC_DIMS]` An array containing the size of each dimension found.
- `das_loc[MAX_NC_VARS]` An index into a table of attributes.

---

2The NetCDF client library has been rewritten a couple of times since these examples were taken from it. Most of the changes involve additional functionality not necessary for the illustrations here. We have also removed some error-checking to make the intention clearer. Therefore the actual code in the NetCDF software may not match these examples. However, these examples will work.
class NCConnect: public Connect {
    private:
        int _ncid;
        int _nvars;
        int _ndims;
        String _dim_name[MAX_NC_DIMS];
        int _dim_size[MAX_NC_DIMS];
        int _das_loc[MAX_NC_VARS];
        void init_list(int i);
        void parse_array_dims();
        void parse_grid_dims();

    public:
        NCConnect(const String &name, const int mode);
        ~NCConnect();

        int &ncid();
        int ndims();
        int nvars();
        int dim_size(const int dimid);
        const String &dim_name(const int dimid);
        int das_loc(const int varid);
        void parse_das_loc();
        void parse_dims();
};

Figure 3.1: The subclass of Connect used with the NetCDF client library.
This additional data allows the client library to emulate most of the NetCDF query functions properly and *locally*.

Now look at the recoded open call of the NetCDF library in figure 3.2. The new function has exactly the same type as the original implementation; it takes the same number and type of arguments and returns the same type. The first operation performed by the new open call is to create a `Connect` object (actually an `NCConnect` object) using the arguments passed to the open call.

After the open call creates the new `NCConnect` object, it makes sure that the user is not trying to open the remote dataset for writing and, if they are not, reads the `DAS` and `DDS` from the dataset. Once read, the `DAS` and `DDS` objects are parsed using `NCConnect`’s member functions `parse_das_loc` and `parse_dims`. These two member functions, along with the additional state variables in `NCConnect`, effectively create the virtual connection. Subsequent calls to the client library for information about the variables (e.g., their size, names, etc.) will be answered using information stored in the symbol table in the `NCConnect` object.

```c
int
ncopen(const char *path, int cmode)
{
    int id;
    NCConnect *c = new NCConnect(path);

    if(cmode != NC_NOWRITE) {
        delete c;
        return(-1);
    }

    c->request_das();
    c->request_dds();

    c->parse_das_loc();
    c->parse_dims();

    return(conns.add_connect(c)) ;
}
```

*Figure 3.2: The recoded open call of NetCDF.*

The code above checks to see whether the user is trying to open a connection for writing. Since DODS is a read-only protocol, this attempt must fail.
This chapter describes how to use the toolkit software to build new client libraries and data servers. Before beginning to build either part of a new DODS application, it is very important to be intimate with the details of the API to be replaced.

To create a client library that can replace the original API implementation at link time means that the client library must present exactly the same interface as the original library. This includes, to the extent that they are widely used, any undocumented features of the original implementation that manifest themselves as symbols that require link-time resolution. Building a client-library requires great understanding of the existing implementation as well as current use of the target API.

To build a good data server for files or data sets encoded using an API it is important to understand the data model(s) the API supports and how they relate to the DODS data models. Each of the various data types that the API supports must be translated into a DODS data type (i.e., one of the DODS classes that descend from BaseType). However, there is often not a one-to-one match between the API’s types and the DODS types. Thus, the data server author must decide how to best translate the API’s types into DODS types so as to preserve as much of the data set author’s intent. This is exacerbated by the use of various conventions that (implicitly) bind several variables together with a data set. When this pattern shows up (as it does with NetCDF) you must decide whether to lump all variables together that appear to use the convention (and thus falsely group some variables) or to group only those which actually are explicitly grouped using whatever the API provides. If you choose the latter then any data sets which follow the convention will lose information. When building the data server it is important to keep such tradeoffs in mind.

The following sections discuss the specifics of building a data server and a client library. The existing NetCDF server and client library are used as examples. Many APIs are very similar in their overall organization. The source code used
for these examples can be found in $(DODS\_ROOT)/src/nc-dods/$. Much of the NetCDF example will be relevant to your task, even if your target API is significantly different. The $(DODS\_ROOT)/src/jg-dods/$ directory contains both a data server and client library for the Joint Geophysical Ocean Flux Study relational data system.
4.1 Data Servers

The DODS data server consists of a *dispatch* program and a set of *filter* programs. The dispatch program reads the incoming URL and decides which of the filter programs to run based on the URL suffix.

A typical DODS data request uses three filters: one to return the *DAS* (.das), one for the *DDS* (.dds), and the third for the data (.dods). A client can also request ASCII data (.asc or .ascii), usage information about the server (.info), or version information about the server and the data (.ver).

The task of building a DODS server can then be separated into the following steps:

1. Create concrete classes of the entire *BaseType* hierarchy, with `read` functions for each data type. Certain APIs cannot handle certain DODS types. For these types, there must still be a concrete class, but it can have a `read` method with a null body.

2. Write functions that use the native API to extract from the dataset the information needed to build the DODS DAS and DDS objects, and then build them with the methods those classes provide.

   **NOTE:** This step has nothing at all to do with DODS. This is between you and your data. DODS makes no demands on how these structures are created. That is, for example, if all the data to be served has the same DDS, feel free to cheat. The only thing that is important is that the structures accurately reflect the relationships of the data.

3. Create filter programs to return the DAS, DDS, data, and server usage and version information.

4. Create a dispatch program to parse an incoming URL and invoke the correct filter program.

To install the finished server, put the filter programs into a web server’s CGI directory, and put the datasets to be served somewhere they can be seen by those filter programs. Refer to the *The OPeNDAP User Guide* for more details about installing a server.

### 4.1.1 The Dispatch CGI

The DODS dispatch CGI program receives a data request from the DODS client, and dispatches the request to one of several filter programs. The dispatch CGI is
stored in a CGI directory on the host machine. Its name is an important detail of its operation. The name should begin with nph-, and end with the letters that distinguish data files containing data formatted with that API from other files. So, for example, NetCDF data files are called foo.nc, so the NetCDF dispatch CGI is called nph-nc.

The dispatch CGI’s job is to parse the incoming URL and execute the appropriate filter programs with the arguments enclosed in the URL. The dispatch CGI is also be responsible for the first level of error information that must be returned to the user. These tasks are easily accomplished in any scripting language. On the off chance you wish to use Perl, DODS provides a Perl class designed to make writing the CGI a simple task.

The file DODS_Dispatch.pm contains the definitions of the DODS_Dispatch class. This class provides several methods used to parse the incoming URL, and one method for delivering error messages to the client. The DODS_Dispatch provides the following methods:

command() Returns the command string implied by the input URL. The command string looks like:

\textit{command filename} -e \textit{query-string}.

Where \textit{command} is the DODS filter program to be run, \textit{filename} is the absolute filename of the dataset on which to run it, and \textit{query-string} is the constraint expression that was enclosed in the URL. Of the DODS_dispatch methods, many dispatch CGI scripts may only need to use this one and print_error_msg. See figure 4.1

query() Returns the query string from the URL. This is the DODS constraint expression.

filename() Returns the absolute filename corresponding to the requested dataset.

extension() Returns the extension on the end of the URL. For DODS, this will be das, dds, dods, info, or ver.

cgi-dir() Returns the absolute pathname of the directory in which the dispatch CGI is stored. This is generally the same as the directory in which the DODS filter programs are stored.

script() Returns the name of the dispatch CGI, minus the nph-, and any suffixes used for a secure server.

\footnote{The nph- is a relic, dating from the misty dawn of the World Wide Web and the first http standards. It stands for “Non-Parsing Header” (See the CGI 1.1 Standard for more information.), and is the only way to pass data through many httpd servers unparsed.}
print_error_message(\texttt{ver})\ This returns an error message to the client, explaining how to use the server. The \texttt{ver} argument should be a string containing the version of the server software. The error message returned is encoded in the \texttt{DODS\_Dispatch.pm} file.

\texttt{print\_help\_message()}\ This returns a help message to the client. This can be issued in response to a confusing or inadequate URL. The help message returned is encoded in the \texttt{DODS\_Dispatch.pm} file.

A sample (simple) DODS dispatch CGI is shown in figure 4.1. This is a Perl script using the \texttt{DODS\_Dispatch} methods. This script assumes that all data is rooted in the http document directory subtree.2

\begin{verbatim}
#!/usr/local/bin/perl
use Env;
use DODS\_Dispatch;

$dispatch = new DODS\_Dispatch;

$command = $dispatch->command();

if ($command ne "") { # if no error...
    exec($command);
} else {
    my $script_rev = '$Revision: 1.6 $';
    $script_rev = s@\$([A-z]*): (.*) $@$2@;

    $dispatch->print_error_msg($script_rev);
}
\end{verbatim}

Figure 4.1: A simple DODS data server dispatch CGI.

\subsection*{4.1.2 The DAS and DDS filter programs}

The simplest way to learn about creating a new filter program to return a dataset’s DAS or DDS is to examine the existing filter programs. In this section, we will examine the NetCDF servers.

The source code for the DAS filter program distributed with the NetCDF server software is shown in figure 4.2. The DAS and DDS filters are very similar, so only the DAS filter will be discussed here. The important differences between the two will be pointed out.

\footnote{You can use this even if you want to access files outside that subtree. Simply use a symbolic link and make sure that your server is set to follow symbolic links.}
The CGI dispatch program makes heavy use of commonly used functions collected in the DODSDispatch class. In the same way, the DODSFilter class collects several commonly used functions for the construction of filter programs. The example program uses several methods of that class. Other useful utility functions are in the cgi-util collection.

The filter program in figure 4.2 can be separated into the following steps:

**line 16** Step 1: The DODSFilter class provides a constructor that parses the argument list to create the data. You can use the OK method to check that the list was parsed properly. Any errors here indicate a mistake in the dispatch CGI itself. This is why the print_usage function prints its message to the WWW server log file when it returns an error object to the client.

**line 21** Step 2: If the user has only requested version information from the server, it is provided here.

**line 26** Step 3: The read_variables function performs the real work of this program. This involves scanning the dataset itself for data variable attributes and using the DAS method functions to assemble the corresponding DAS. This operation is specific to the data access API in use, so does not make a good example.

**line 29** Step 4: Each of the filter programs must create a Multipurpose Internet Mail Extensions document to hold its return value. The DAS and DDS filters return a text MIME document; they set up the MIME headers using the utility function set_mime_text.

**line 34** Step 5: Once the data set has been read and the attribute table built, the DAS ancillary file is loaded. The example filter looks for a file with the same root name as the data set and an extension of .das. If such a file exists, it is read in using the DAS member function DAS::parse and the information it contains is merged with the DAS built from the dataset.

**line 37** Step 6: Finally the DAS member function print is used to send the textual representation of the DAS to the client. When it is invoked by the httpd daemon, the dispatch CGI’s standard input and output are a socket connected to the remote client process. This means that since the filter is invoked by the dispatch script, its output goes directly to the client. The DODSFilter send_das method looks something like this:

```cpp
DODSFilter::send_das(DAS &das)
{
    set_mime_text(dods_das);
    das.print();

    return true;
}
```
4.1 Data Servers

```cpp
#include <iostream.h>

#include "DAS.h"
#include "cgi_util.h"
#include "DODSFilter.h"

extern bool read_variables(DAS &das,
    const char *filename, String *error);

int main(int argc, char *argv[])
{
    DAS das;
    DODSFilter df(argc, argv);

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

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

    String errMsg;
    if(!read_variables(das, df.get_dataset_name(), &errMsg)){
        Error e(no_such_file, errMsg);
        set_mime_text(dods_error);
        e.print();
        return 1;
    }

    if (!df.read_ancillary_das(das))
        return 1;

    if (!df.send_das(das))
        return 1;

    return 0;
}
```

Figure 4.2: The DAS filter program.
Note that the example filter in figure 4.2 does not use any caching. It is possible to build a more sophisticated filter program that saves the generated DAS to a text file and then uses that file without first interrogating the data set, thus saving on access. It is also possible to write a DAS by hand and *always* use that if the data set does not contain any of the type of information that the DAS has.
Caching 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 filter programs use a text representation for transmission from the server to the client, it is simple 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 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 dataset 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 possible to read text version of the DAS/DDS object, and then read only the new parts of the data set. The binary object will be added to as needed.

**NOTE:** The DAS/DDS software may not properly update changed data (data that was present in a previous version of the data set, but is now different) nor is it straightforward to remove data which is no longer present in the data set. In these cases it is usually better to regenerate the DAS/DDS from scratch.

4.1.3 The Data filter

The data filter program is structured similarly to both the DAS and DDS filters except that it returns a binary MIME document rather than text and that it takes two arguments instead of just one. In addition to the data set or file name (argument 1) it also takes the DODS constraint expression (argument 2, which was enclosed in the URL’s query).

The NetCDF data filter is all but identical to the DDS filter. The only difference is that it calls the `send_data` method of `DODSFilter` to send the binary data over the network. This function calls the `DDS send` method.

If for some reason you cannot use the `send` member function of DDS, then you must ensure that the the `read`, `CE evaluation` and the `serialize` operations are all carried out in the correct order. Furthermore, you must ensure that the return value of the data filter is a binary MIME document with a text prefix (currently, DODS
does not use the multi-part MIME standard); that is a regular binary MIME
document with a section at the start that is text. This text is the DDS
generated after evaluating the projection clauses of the constraint expression. The text part is
separated from the data by the keyword “Data:” at the start of the line.3

The ASCII Data Filter

DODS is packaged with a filter to translate a DODS data stream into an ASCII
data file. Clients can request ASCII data by appending .asc or .ascii to their
URL instead of .dods. The asciival program is useful as a standalone client
(see The OPeNDAP User Guide), but may also be used by a server to provide
ASCII data.

A request for ASCII data is processed as any other request for data, but the final
output of the data filter is piped into the asciival program and the result
returned to the client:

nc_dods Data.nc | asciival -m -- -

The DODS_Dispatch class takes care of this step automatically, when it
encounters a request using .asc or .ascii.

4.1.4 The Usage Filter

Client requests containing a .info suffix should return to the client HTML text
containing documentation of both the server usage and the dataset named in the
query. DODS provides a usage filter that can be used for this purpose. The
DODS_Dispatch class invokes this filter.

The DODS-provided usage filter accepts two arguments, the data file name
requested and the name of the CGI script (the dispatch CGI) in use:

usage filename CGI-name

The usage filter looks in the dataset directory for a file called filename.html, and
in the directory specified in the CGI-name argument for a file called
CGI-name.html. These two files must contain HTML, but without the <html>,
<head>, or <body> tags.

For example, suppose a dispatch CGI using the DODS_Dispatch class receives a
URL like this:

http://dods/cgi-bin/nph-nc/data.info

In this case, the usage filter looks for two files: cgi-bin/nph-nc.html and
data.html (the htdocs directory is assumed in the second case). The contents of

3The “Data:” keyword is not in the scope of the text DDS so it is possible to have the text Data:
in the DDS.
these two files are concatenated with an HTML representation of the DAS and DDS for the data.nc file, and the whole thing is returned to the client. If the HTML files are not found, the returned document contains only the DAS and DDS.

4.1.5 Documenting Your Work

If you do write a server, and intend to circulate it beyond your own site, here are some guidelines for documenting that server that will help others use it.

Since there are two sets of “users” for a data server program, there are two sets of instructions that need to be prepared for a given server. One set will be read by the person who installs and maintains the server on the host platform. The other set is designed to be read by people who intend to request data from that server. These users will get this documentation by submitting queries to the Info Service, in rather the same way that many UNIX commands have a -usage option.

In addition to these two documents, all servers should include a set of text files in their distribution directory.

**The README File**

The README file should contain the following information:

- A brief overview that describes the purpose and method of operation of the server.
- The revision level of the server.
- Any features the local httpd daemon must support to use this server.
- Any data translations that this data server can do. If any are done, they should be described in detail, so that users can know what data they get.

**The ERRORS File**

The ERRORS file should contain a complete list of the error messages and explanations that might ever be issued by the server.

**Installation Notes**

These instructions should be included in a file called INSTALL which is to be included with the server distribution. At a minimum, they should cover the following topics:
Configuring and compiling the server code. Ideally, there should be a configure script included, but detailed instructions on editing the Makefile will often suffice. Remember to install the usage data file somewhere the server can find it.

Are there any environment variables that must be defined in order to run the server? Are there other programs (e.g. gzip that must be installed on the host machine?

What configuration options are there for the installed server? This covers issues like enabling data compression, ancillary data caching, and choosing the GUI manager program with which the server will communicate. If there are performance trade-offs associated with each option, note them here.

Ancillary data files:

Must the installer prepare ancillary data files by hand, or are these created automatically and cached?

If they must be created, where ought they be put?

If they are cached, where are they kept?

Also, if the ancillary data files are cached, what implications are there for updating the data sets served by this server? (i.e. must the ancillary data files be updated also? Deleted and recreated?)

What temporary files will be created by the server? Where will they be stored? Under what conditions may (or must they) they be erased?

Information Files

The information files contain the information that remote users of this server will use to figure out how to use this server and its datasets once it is installed somewhere. The files are used in constructing the HTML page for the info server. The .info results can include information about both the server and the current dataset. (In fact, the results will usually include the DAS and DDS of the dataset named in the URL.)

When a user appends .info to a URL, the info service is activated. This service collates information about the server and the dataset (from the DAS, DDS, lists of global attributes, and variable summaries), and assembles that information in an HTML document. The server then looks for additional HTML files created by the server’s administrator, and appends them to the original file, and returns the whole document to the client.

Although it is possible merely to rely on the collated data to describe a server, we hope that server writers will provide rich, human-friendly descriptions of the server’s usage and the accompanying datasets. These files can be thought of as “usage” or “README” files. At a minimum, they should cover:
Any special data functions defined by the server that can be used in a constraint expression, and

Any data model translations the server supports, and how they are to be controlled by the user.

A list of the programs a user should have to use certain features of the server. For example, note here that the server expects that the GUI manager is running a Tcl interpreter.

A list of the error messages that the user is apt to see. Include explanations of the conditions that may have caused them, and any steps the user may take to recover from them.

The answers to any questions you are frequently asked about this server or its usage.

The usage data file need not be any more elaborate than any man page.

To create information for a server, write an HTML fragment using the format given below, and put the HTML file in the same directory as the server. Name it using the base name of the server; for example, the HTML file that describes the netCDF server (made up of nph-nc, and ncdas, ncdods and so on) is called nc.html.

This example shows the correct HTML tagging for server information:

```html
<h3>Server Function:</h3>
<dl>
  <dt>geolocate(variable, lat1, lat2, lon1, lon2)</dt>
  <dd>Returns the elements of <em>variable</em> that fall within the box created by (<em>lat1</em>,<em>lon1</em>) and (<em>lat2</em>,<em>lon2</em>). </dd>
  <dt>time(variable, start_time, stop_time)</dt>
  <dd>Returns the elements of <em>variable</em> that fall within the time interval <em>start_time</em> and <em>stop_time</em>. </dd>
</dl>
```

For datasets, put the HTML file, tagged using the format given below, in the same directory as the datasets. Name it using the base name of the datasets; for

---

4Remember that the “how” is to be answered very specifically, and on the user’s level (i.e. “Do such-and-such, spelled like this, to make the array returned be nx5 instead of 5xn.”), and not on the programmer’s level (i.e. “You use the invert method to return an array of 5xn instead of nx5.”)
example, the HTML file for fnoc1.nc, fnoc2.nc, and fnoc3.nc might be called fnoc.html. This example shows the correct HTML for a dataset information file:

```html
<h3>
About the dataset
</h3>
This is where the server administrator would supply information about the dataset. And so on...
<p>

You may prefer to override this method of creating documentation and simply provide a single, complete HTML document that contains general information for the server or for a group of datasets. For example, to force the info server to return a particular HTML document for all its datasets, you would create a complete HTML document and give it the name dataset.ovr, where dataset is the dataset name. The HTML file in this case would look like this:

```html
<html>
<head>
<title>Override document</title>
</head>
<body>
<h2>
Test dataset
</h2>
This is where the server administrator would supply information about the dataset(s) and what-have-you.
</body>
</html>

Remember to ensure that the installation instructions cover installing the usage data file in a place where the server can find it.
4.2 Client Libraries

The goal of building a client library is to provide a drop-in replacement for an existing API so that user programs written for that API can switch to the DODS version and access remote DODS data. The user programs should not require any modification to change over to the DODS client library version of the API. However, the API will clearly need substantial changes to its current implementation.

In order to build the DODS client library for a particular API, it is useful to divide the API to be re-implemented into five categories of functions:

>>> Open or connect

>>> Variable information read

>>> Data read

>>> Write

>>> Close or disconnect

4.2.1 Rewriting the Open and Close Functions

The functions that perform the dataset “open” and “close” operations must be implemented so that information about the data set can be retrieved from the data server. These 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, be the dataset’s DAS and DDS.

The open function for a DODS client library version of a given API 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 also be accessed through the original API. In any case, the distinction of local or remote is made on the basis whether a URL is used to reference the data object, or a local filename.

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. This is the Connect class object. Once this 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. You add this data to the Connect objects when you sub-class them for a particular API. Subsequent accesses to the data set will include this identifier (or pointer), and each function that 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 should use the state information accessible with the file identifier or pointer returned by the open function to determine if the dataset is local or remote. 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. You can do this by destroying the Connect object.

See Section 3.2 on page 26 for an example of a recoded open function and a description of its use. (The example uses the NetCDF API.)

4.2.2 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 attribute information about the variables that is more often than not intended 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.

While many ‘self-describing’ APIs may have dozens 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 stored state information (DAS and DDS) to answer the request for data.

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.

4.2.3 Reading the Values of Variables from a Dataset

To read data values from a dataset using a typical data access API, a user would submit to some API function the name of the variable to be read. The DODS client library version of this same function must take that variable name and use it to construct a constraint expression. (See Section 4.3.2 for more information on using constraint expressions to access data.) The constraint expression must then be appended to the dataset URL (with the suffix .dods), and the resulting URL sent out into the internet.

For example, to get a variable called var from a dataset at:

http://blah/cgi-bin/nph-nc/weekly.nc.dods

you would use the URL:

http://blah/cgi-bin/nph-nc/weekly.nc.dods?var
The Connect class contains a member function, request_data that performs this task. It takes the constraint expression and the suffix to use for requesting data, appends them to the Connect URL, and sends the entire string off to retrieve its corresponding data.

The request_data function returns a pointer to a DDS object, which contains the data as well as the structure description corresponding to the data request.

Once the request_data member function has returned, the client library must still call the deserialize member function (which is part of the DODS Type Classes) for each returned variable. The client library should use the variable objects contained in the DDS object returned by request_data to invoke the deserialize member function. Once that is done, the data values are stored in the internal buffers of the variable objects in the new DDS. The client library should store this new DDS, along with the constraint expression passed to request_data so that future requests by the user program for the same information can be handled without accessing the remote data server.

The data values of variables in a DDS are accessed using the buf2val member function for the cardinal and vector types and by accessing the values of fields for constructor types.

Translation

For a DODS client library to be robust, it may have to be equipped to deal with data types it was not designed to use. For example, the NetCDF software cannot manipulate a DODS Sequence. But a user can use the DODS version of the NetCDF library to request data from a server that provides Sequence data. When cases like this arise (and they arise fairly often), the author of the client library must choose an appropriate data type into which the served data is to be translated, and implement functions to do that translation.

Often, translation from one data type to another is a simple task. Translating an Array into Sequence format is fairly straightforward, although there are several ways to do it. (The author of the client library should choose one, and document that choice in a README file.)

Other translations are more complex, and may even require that the client library violate the semantics of the original API, or of one of the DODS data types. For example, translating a Sequence to an Array in NetCDF requires that the client know in advance the length of the Sequence, which is not necessarily known.

For the Sequence data type, the DDS contains only the current instance of the data. Repeated calls to the Sequence’s deserialize function are required to return successive instances of the sequence.
4.2.4 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 would be operationally much more complex than DODS. 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.

4.2.5 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 files it is necessary to add software to 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 because both libraries 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, you must 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.
4.3 Using Constraints

Constraint expressions are an important part of DODS, providing a powerful way to control how data is accessed without forcing the Data Access Protocol to support a lot of different messages. Constraint expressions are used to select which variables will be extracted from a data set by both the user and by the client library. The constraint expression syntax is described in detail in the The OPeNDAP User Guide.

4.3.1 How Constraint Expressions are Evaluated

The server-side constraint expressions are evaluated using a two step process. Every constraint expression has two parts, the projection and the selection subexpressions. The projection part of a constraint expression tells which variables to include in any return document describing the data set and the selection subexpression limits the returned data to variables with values that satisfy a set of relational expressions. The projection subexpression is evaluated when the entire constraint expression is parsed; at parse-time the server’s copy of the data set’s DDS is marked with the variables included in the projection. The selection subexpression, however, is not evaluated until values are read from the data set. One way to classify the projection and selection subexpressions is that projections depend solely on the logical structure of a data set, while selections depend on the values of particular variables within that data set.

4.3.2 Different Ways of Using Constraint Expressions

There are two different ways that constraint expressions can be used. One is by the client library and the other is by the user. When writing a client library that has features for selecting variables or parts of variables, try to code the replacements to those calls so that they build up DODS constraint expressions that will request only the data the user wants. Then read the data from the returned DDS and store it in the variable(s) passed to the API call by the user. This is a much better solution than requesting the entire variable from the data set and then throwing away parts of it.

Suppose that the user program (via the APIs functional interface) asks for the data in variable \( X \). The constraint expression that will retrieve \( X \) is simply ‘\( X \)’. Suppose, given the following DDS that the user program requests the two variables \( u \) and \( v \) from the embedded structure.
Using the Toolkit

Dataset {
    Int32 u[time_a = 16][lat = 17][lon = 21];
    Int32 v[time_a = 16][lat = 17][lon = 21];
    Float64 lat[lat = 17];
    Float64 lon[lon = 21];
    Float64 time[time = 16];
} fnoc1;

A constraint expression that would project just those variables would be fnoc1.u,fnoc1.v. To restrict the arrays \(u\) and \(v\) to only the first two dimensions (time and lat), the projection subexpression would be:

\[
\text{fnoc1.u}[0:15][0:16], \text{fnoc1.v}[0:15][0:16]
\]

Both of these constraint expressions have null selection subexpressions. Note that the comma operator separates the two clauses of the projection subexpression. Also note that whitespace is ignored by the constraint expression parser. See the grammar for CEs in the *The OPeNDAP User Guide* for more information about constraint expression grammar and the kind of things that can be done with the projection subexpression.

The user program may have an interface that provides the user with a way to request only certain values be returned. This is particularly true for APIs such as JGOFS that support access to relational data sets. Suppose the following DDS describes a relational data set:

Dataset {
    Sequence {
        Int32 id;
        Float64 lat;
        Float64 lon;
        Sequence {
            Float64 depth;
            Float64 temperature;
        } xbt;
    } site;
} cruise;

To request data with a certain range of latitude and longitude values, you can use a selection subexpression like this:

\[
\& \text{lat} \geq 10.0 \& \text{lat} \leq 20.0 \& \text{long} \geq 5.5 \& \text{long} \leq 7.5
\]

Note that each clause of the selection subexpression begins with a \(\&\) and that the clauses are combined using a boolean \(\&\) and. Finally, using the previous DDS, if a user requested only depth and temperature given the above latitude and longitude range (i.e., the user program requests that only the depth and temperature values be returned given a certain latitude and longitude range) the client library would use the following constraint expression:

\[
\text{site.xbt.depth, site.xbt.temp \& lat} \geq 10.0 \& \text{lat} \leq 20.0 \& \text{long} \geq 5.5 \& \text{long} \leq 7.5
\]
A second way that constraint expressions can be used is that users may specify an initial URL with a constraint expression already attached. In this case the request_data member function will append the constraint expression built by the client library to the one supplied by the user and request data constrained by both expressions. From the standpoint of a client library (or a data server, for that matter) there is no difference between a URL supplied with an initial constraint and one supplied without one.
To link a user program to the DODS client-library version of a data access API library, you need only substitute the name of the reimplemented API library for the original, and add the DODS Data Access Protocol library (libdap++) to the link list.

Because the DAP library and the API have circular dependencies they must each be included on the linker command line twice. An example of this can be seen in the DODS-NetCDF Makefile; the value of LIBS is passed to the linker ld. Note that the API library is listed first.

```
LIBS = -lnc-dods -ldap++ -lnc-dods -ldap++
```

You should have users link their programs using gcc or g++ since the libraries are all built using those tools. In particular, g++ includes libstdc++ (and libg++) by default when it builds an executable program from object modules. If you use gcc instead of g++ when you link, be sure to include these libraries as well after all the libraries listed above. If you don’t use gcc, but instead use the linker directly (i.e., you call ld yourself) you are on your own - you can use gcc -V to determine what flags and additional libraries it uses that are specific to your system and then experiment with those. We cannot tell you how to proceed since each UNIX variant requires different flags and libraries.
The html version of *The DODS Toolkit Reference* is a separate document. Please refer there for detailed information about the DAP classes.
This appendix describes in functional terms the operation of a server.

Note that descriptions of server outputs naturally and unavoidably refer to certain kinds of inputs and vice versa. This makes it difficult to create definitions without forward and backward cross-references. Please be prepared to read this specification with a thumb between the pages as you flip back and forth from the output to the input sections.
A.1 Outputs

A server sends to a client one of three different sorts of messages:

- HTML;
- ASCII data; or
- Binary data.

A.1.1 HTML Data

There are three different kinds of HTML data returned by a DODS server. Clients that request these data should be prepared to display the HTML to the user.

Server Information (usage) One kind of HTML data is returned in response to a request for server information (using the .info URL suffix), and contains usage information for the server, including a formatted version of the dataset DAS and DDS. This information is formatted by the usage service program, invoked by the server.

WWW Interface The forms-based DODS WWW interface returns HTML data to the client. This can either be a form a user can use to create a DODS constraint expression or a DODS directory listing, depending on whether the URL indicates a file with the .html suffix, or a directory containing other files.

Error messages A badly formed URL will result in a DODS error message, which is simply some HTML text describing the supported URL suffixes. (See DODS_Dispenser.pm.) Note that though an error message could in theory be returned to any client, whether or not they can display HTML, in practice, only web browser clients are prone to these kinds of errors. Aside from web browser clients (e.g. Netscape), the DODS clients issue their server requests through the DODS client core libraries, which format the URL according to the DODS conventions.

NOTE: There are two kinds of DODS error messages. The particular one described above is issued in response to a badly formed URL. Other kinds of errors are returned as ASCII data within a DODS data document. See Section A.1.3 on page 57.
A.1 Outputs

A.1.2 ASCII Data (Text)

These are the different kinds of ASCII data returned by a fully-equipped DODS server. (“Fully equipped” implies that the server has, on its execution path, all the supported DODS service programs.)

**DDS** The DODS Data Descriptor Structure is a description of the data contained in the dataset. It contains information about the data types and names represented in the dataset. Programmers can think of the DDS as containing the type declarations for the data. The DDS is fully described in *The OPeNDAP User Guide*. The DDS is created with a service program called `*_.dds`, where the `*` is the same two-letter abbreviation used in the server name (nph-*).

**DAS** The DODS Data Attribute Structure contains information about the data in the dataset—the metadata. It is a hierarchical list of name-type-value triples, where the names of the containers correspond to entries in the DDS. The DAS is fully described in *The OPeNDAP User Guide*. The DAS is created with a service program called `*_.das`, where the `*` is the same two-letter abbreviation used in the server name (nph-*).

**ASCII Data** If a DODS server is equipped with a service program called `asciival`, it can convert binary data to ascii data on the fly, allowing you to use a standard web browser (such as Netscape) to examine data. This feature can also be used to import data into a client that may not be able to process DODS data encoded in the standard binary format.

A.1.3 Binary Data

The DODS data document consists of two parts, a DDS describing the data returned, and the data itself. It may also consist of ASCII data describing an error condition.

**DDS** The DDS returned with the data differs slightly from the DDS returned by a simple request using the `.dds` URL suffix. Whereas the DDS returned in response to that URL is a description of the dataset available, the DDS returned in a data document describes the data being returned. If you ask for an entire dataset, there will be no difference between these two DDS's. However, if you request only a part of a dataset, the DDS included in the data document will only reflect the part of the dataset returned to you.

As an example, consider a dataset containing an array with one hundred elements. The DDS for this dataset will contain an array with one hundred elements. However, if you send a server a request for just the first fifty elements of the array, the DODS data document returned will contain a DDS with only fifty elements. This is illustrated in figure A.1.
Data  The data requested from a server is (unsurprisingly) also contained in the
DODS server’s response to a data request. The data is encoded using the
XDR standard (eXternal Data Representation), and packed into the second
part of the response document. For more information on XDR, see Internet
RFC 1014. For further details, see Section A.1.3 on page 59, below.

Error Messages  A DODS data document may contain an error message. This is
a DODS Error object (containing an ASCII error message and some other
data) stuck into the HTTP document where the data would usually go. This
is where error conditions on the server are noted, as well as badly formed
constraint expressions and file names.

The HTTP response containing the DODS data document is formatted like this:

HTTP/1.0 200 OK
XDODS-Server: <server/dap version string>
Content-type: application/octet-stream
Content-Description: dods-data
Content-Encoding: deflate
<blank line>
<DDS>
Data:
<binary data, each variable given in the order it is listed
in the DDS>
<EOF>

Note the following:

◆ DODS servers use HTTP 1.0.

◆ The version string (after the XDODS-Server: header) must be
  \text{ ¡text\¡/¡version number\¡} where version number is x.y or x.z.y. This version
  number is parsed on the client side to ensure that a client is communicating
  with a compatible server.

◆ The Content-Encoding is used only when the document is compressed
  using ‘deflate’. (This is the same as the compression used by the gzip
  program, and is implemented in libz.a.)

◆ Content-Description is used by the DODS client to figure out if the
  object is an error instead of a data object. If it is, the
  Content-Description field should read dods-error.
Encoding the DAP Data Types

The DODS transmission protocol separates variables into three classes: scalars, arrays, and sequences. A scalar is included in the DODS data document simply by writing its XDR representation. An array is sent by writing the number of elements (as an XDR int) followed by the elements themselves, also in the XDR format. However, arrays of Byte, Int16, Int32, UInt16, UInt32, Float32, and Float64 actually have their size sent twice, once by the DAP software and once by the XDR software. Here’s a hex dump of some Int32 array data that shows this behavior:

The length bytes for the \texttt{\{data\}} section start at address 0x45 (The length is 0x00000015.) and is repeated at 0x49. Here’s how you can see this:

geturl "http://dods.gso.uri.edu/cgi-bin/nph-nc/data/fnoc1.nc?u[0:0][0:0][0:20]"

The DDS for this dataset is (use \texttt{geturl -d}):

```plaintext
Dataset {
    Int32 u\{time_a = 16\}[lat = 17][lon = 21];
    Int32 v\{time_a = 16\}[lat = 17][lon = 21];
    Float64 lat[lat = 17];
    Float64 lon[lon = 21];
    Float64 time[time = 16];
} fnoc1;
```

Structures and Grids are sent by serializing their components, one by one, in the order of their declaration in the DDS.

Sequences have a much more complex encoding scheme because one sequence may contain another sequence \textit{and} because the sequence type provides no information about the length of a given instance. To send sequences the DAP uses two different algorithms, one up to DODS version 2.14 servers and clients and a better (more compact one) after that. DODS clients at version 2.15 and later can all read from both servers \textit{but} they assume that a server that does not announce its version is very old and will attempt to read Sequences using the old algorithm. (So if you’re writing a new DODS server that serves Sequences, you must remember to address the version requirement.)

Here’s how the new algorithm encodes a sequence:
Serialize row of the sequence by:

1. Writing the start of instance marker (0x5A), and
2. Serializing (writing the XDR encoding of) each element in the row in the order of appearance, then

3. Write the end of sequence marker (0x5A).
A.2 Inputs

The input to a DODS server is contained in an HTTP “GET” request. Unlike a POST, the information in this kind of request is all in the URL. Consequently, examining the parts of a DODS URL will illustrate all the different sorts of requests a DODS server can handle.

Figure A.2 contains a description of the parts of a DODS URL, not including the “Constraint Expression.” The constraint expression and the parts of the DODS URL are described in detail in *The OPeNDAP User Guide* and *The OPeNDAP User Guide*.

### A.2.1 Request types

A DODS server is equipped to respond to several different request types. Each request type is signified by a different URL suffix. The server itself is just a dispatch script, that determines the type of a request, and dispatches the request to the appropriate service program, and relays its result back to the client.

In figure A.3, a DODS client makes a GET request to a DODS server (which is just an httpd daemon equipped with a bunch of CGI programs). The daemon invokes the DODS Server, which is a simple-minded dispatch script which in turn invokes the DAS, DDS, Data, or other service program.

### Table A.1: Table of URL suffixes.

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Service Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.dds</td>
<td>*_dds</td>
<td>Returns the Data Descriptor Structure for the specified dataset.</td>
</tr>
<tr>
<td>.das</td>
<td>*_das</td>
<td>Returns the Data Attribute Structure for the specified dataset.</td>
</tr>
<tr>
<td>.dods</td>
<td>*_dods</td>
<td>Returns binary data in the form of a DODS data document. See Section A.1.3 on page 57.</td>
</tr>
<tr>
<td>.asc</td>
<td>asciival</td>
<td>Converts data requests to ASCII values before sending them back to the client. This service is useful for invoking from simple web browsers.</td>
</tr>
<tr>
<td>.info</td>
<td>usage</td>
<td>Returns an HTML formatted version of the dataset DDS and DAS, and any other server and dataset information provided in *.ovr files.</td>
</tr>
<tr>
<td>.html</td>
<td>none</td>
<td>Returns a URL constraint expression builder form, based on the dataset DDS and DAS. This is the DODS WWW interface.</td>
</tr>
<tr>
<td>.ver</td>
<td>none</td>
<td>Returns the server version information.</td>
</tr>
</tbody>
</table>
The request types, their suffixes, and the service programs are listed in table A.1.

### A.2.2 Constraint expressions

A DODS server can accept a “constraint expression” contained in the URL query string. The DODS constraint expression describes how a DODS server should subsample a DODS dataset before sending the data back to the client. The details of the constraint expression syntax are covered in *The OPeNDAP User Guide*. What’s important here is simply that the constraint expression is a logical expression with two clauses: projection and selection.

The “projection” clause of a constraint expression specifies the data variables requested by the client, and the “selection” clause specifies the condition under which the client wants them. That is, the projection clause might specify that the client wants to see oceanic temperature data, and the selection clause would specify that only records from below 1000 meters should be returned.

### A.2.3 Server functions

Within the context of a constraint expression, a server can implement functions a client would use to specify data. Since the constraint expression has two kinds of clauses, there are two kinds of server functions: projection and selection.

To implement another server-side constraint function, see . Following is a list of the canonical server-side functions implemented in all DODS servers.

**Selection**

yes

**Projection**

yes
Figure A.1: The DODS Data Document and the DDS. For the dataset containing the vector `temp`, with 100 elements, the top "Simple DDS Request" shows what the DDS might look like for that dataset. The bottom "DODS Data Document" shows what might be returned by a request for all the even-numbered elements of the `temp` array. Note that the DDS has been altered to allow for the reduced number of elements in the returned data array.
Figure A.2: Parts of a DODS URL (without a constraint expression)
A DODS client is any web client that makes a service request to a DODS server. Since several of the DODS services return ASCII and HTML data, any web browser, such as Netscape Navigator can be considered a DODS client, so long as it is in the process of making a suitable request to a DODS server. The clients of interest in this appendix, however, are clients that use the DODS DAP (Data Access Protocol) library to make their requests for data.

Of these clients, there are two varieties: clients that have been written expressly for DODS, and clients that existed in some form already, and that have been adapted to use with a DODS client library.
Figure B.1: The Original Program, untouched by DODS. The application's code accesses data by calls to the netCDF library functions, linked with the program. Data access is direct, with the application program accessing local disk files to read data.
Figure B.2: The Modified Program, using the DODS netCDF client library. The application's code now accesses data by calls to the DODS netCDF library functions. These are written to be functionally identical to the original netCDF functions, but instead of using a local disk to retrieve data, this library invokes functions from the DODS DAP library, which makes HTTP GET requests to a DODS server. The client code is unchanged.
Figure B.3: Another Way. An application program can also call the DODS DAP directly, eliminating the need for a client library. When starting from scratch, this is probably easiest, unless you are an old hand at one of the supported data access APIs.
# Index

## Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>.asc(ii), see (sc)</td>
<td>38</td>
</tr>
<tr>
<td>.info, see (nfo)</td>
<td>38</td>
</tr>
<tr>
<td>.ver, see (er)</td>
<td>31</td>
</tr>
<tr>
<td>ancillary data</td>
<td>3</td>
</tr>
<tr>
<td>API</td>
<td></td>
</tr>
<tr>
<td>implementing</td>
<td>29</td>
</tr>
<tr>
<td>application</td>
<td></td>
</tr>
<tr>
<td>linking</td>
<td>51</td>
</tr>
<tr>
<td>Array class</td>
<td>8</td>
</tr>
<tr>
<td>asc</td>
<td></td>
</tr>
<tr>
<td>URL suffix</td>
<td>38</td>
</tr>
<tr>
<td>ASCII</td>
<td></td>
</tr>
<tr>
<td>CGI</td>
<td>38</td>
</tr>
<tr>
<td>ascii, see (sc)</td>
<td>38</td>
</tr>
<tr>
<td>ASCII data</td>
<td>38</td>
</tr>
<tr>
<td>asciiival</td>
<td>38</td>
</tr>
<tr>
<td>attribute</td>
<td></td>
</tr>
<tr>
<td>table</td>
<td>5</td>
</tr>
<tr>
<td>attribute data</td>
<td>3</td>
</tr>
<tr>
<td>attribute information</td>
<td>4</td>
</tr>
<tr>
<td>attribute tables</td>
<td>5</td>
</tr>
<tr>
<td>AttrTable class</td>
<td>5</td>
</tr>
<tr>
<td>BaseType class</td>
<td>7</td>
</tr>
<tr>
<td>buf2val</td>
<td>45</td>
</tr>
<tr>
<td>Byte class</td>
<td>8</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>cache</td>
<td></td>
</tr>
<tr>
<td>DDS</td>
<td>37</td>
</tr>
<tr>
<td>objects</td>
<td>37</td>
</tr>
<tr>
<td>caching</td>
<td>3</td>
</tr>
<tr>
<td>CGI</td>
<td></td>
</tr>
<tr>
<td>ASCII</td>
<td>38</td>
</tr>
<tr>
<td>data</td>
<td>37</td>
</tr>
<tr>
<td>dispatch</td>
<td>31</td>
</tr>
<tr>
<td>example dispatch</td>
<td>33</td>
</tr>
<tr>
<td>example filter</td>
<td>33</td>
</tr>
<tr>
<td>filter</td>
<td>31</td>
</tr>
<tr>
<td>usage</td>
<td>38</td>
</tr>
<tr>
<td>CGI 1.1 standard</td>
<td>32</td>
</tr>
<tr>
<td>cgi-dir</td>
<td></td>
</tr>
<tr>
<td>DODS_DISPATCH_METHOD</td>
<td>32</td>
</tr>
<tr>
<td>cgi-util functions</td>
<td>34</td>
</tr>
<tr>
<td>class</td>
<td></td>
</tr>
<tr>
<td>Array</td>
<td>8</td>
</tr>
<tr>
<td>BaseType</td>
<td>7</td>
</tr>
<tr>
<td>Byte</td>
<td>8</td>
</tr>
<tr>
<td>Connect</td>
<td>18</td>
</tr>
<tr>
<td>Float32</td>
<td>8</td>
</tr>
<tr>
<td>Float64</td>
<td>8</td>
</tr>
<tr>
<td>Grid</td>
<td>10</td>
</tr>
<tr>
<td>Int16</td>
<td>8</td>
</tr>
<tr>
<td>Int32</td>
<td>8</td>
</tr>
<tr>
<td>List</td>
<td>9</td>
</tr>
<tr>
<td>Sequence</td>
<td>9</td>
</tr>
<tr>
<td>Str</td>
<td>8</td>
</tr>
<tr>
<td>Structure</td>
<td>9</td>
</tr>
<tr>
<td>UInt16</td>
<td>8</td>
</tr>
</tbody>
</table>
INDEX

UME, 8
URL, 8
Vector, 8
client library, 29
  building, 43
close, 43
open, 43
read, 44
variable information, 44
write, 46
client-server connection, 18
close
  client library, 43
command
  DODS_Dispatch method, 32
compound, 7
compound types
  sub-classing, 25
Connect class, 18, 43
request_data, 44
sub-classing, 26
example, 26
connection
  managing, 18
Connections class, 19
constraint expression, 47
evaluation, 47
example, 48
copy function, 22

D
DAS
  diagram, 4
  internal vs. external
    representation, 3
  transmission, 5
DAS class, 3, 44
DAS filter
  CGI, 33
DAS object, 4
data
  CGI, 37
  filter, 37
  usage, 40
Data Attribute Structure, see (AS)3
data attributes, 3
data description, 3
Data Description Structure, see (DS)3
data model, 3, 29
translation, 45
data types
  compound, 9
  DODS, 7
  simple, 8
  vector, 8
DDS
  cache, 37
  internal vs. external
    representation, 3
DDS class, 3, 44
DDS filter
  CGI, 33
DDS object, 6
diagram, 6
deserialize, 45
dispatch, 31
  CGI, 31
  CGI example, 33
dispatch CGI, 31
documentation
  how to write, 39
DODS
  toolkit contents, 1
DODS types, 7
  compound, 9
  simple, 8
  using, 22
  vector, 8
DODS_Dispatch, 32
DODSFilter Class, 34

E
expression
  constraint, 47
extension
  DODS_Dispatch method, 32

F
filename
DODS_Dispatch method, 32
filter, 3, 31
   CGI example, 33
data, 37
   netCDF, 33
   writing, 33
filter CGI, 31
Float64 class, 8

G
   global attribute, 4
   Grid class, 10

I
   info
      URL suffix, 38
   info service, 40
   installation notes
      how to write, 39
   Int32 class, 8

L
   libraries
      object, 51
   linking
      application, 51
   List class, 9
   local access, 46

M
   map vectors, 10

N
   netCDF filter, 33
   non-parsing header, 32
   nph-, see (on-parsing header)32

O
   object
      DAS, 4
      DDS, 6
   object libraries, 51
   object persistence, 3
   objects
      cache, 37
      open
      client library, 43
      using Connect class, 28

P
   performance improvement, 3
   Perl
      dispatch class, 32
   persistence
      object, 3
   print_error_message
      DODS_Dispatch method, 32
   print_help_message
      DODS_Dispatch method, 33
   projection
      constraint expression, 48
   ptr_duplicate, 22

Q
   query
      DODS_Dispatch method, 32

R
   read
      client library, 44
   read function, 22
   request_data, 44

S
   script
      DODS_Dispatch method, 32
   selection
      constraint expression, 48
   Sequence
      translation, 45
   Sequence class, 9
   server
      building, 31
      documentation
         how to write, 39
Service
  Usage, 41
simple, 7
simple types
  sub-classing, 23
state, 10
Str class, 8
Structure class, 9
sub-classing
  compound types, 25
  Connect, 26
  example, 23
  simple types, 23
  type classes, 22
  vector types, 24
suffix
  .info, 38

T
  toolkit
    contents, 1
  translation
    data model, 29, 45
Type Hierarchy, 7
typographic conventions, v

U
  UInt32 class, 8
  URL class, 8
  URL suffix, 31
    asc, 38
    info, 38
    ver, 31
usage
  CGI, 38
usage data
  how to write, 40
usage service, 40

V
  val2buf, 24
variable information
  client library, 44
Vector class, 8

vector types
  sub-classing, 24
ver
  URL suffix, 31
virtual connection, 3

W
  write
    client library, 46
  writing
    filter, 33