DODS User Guide
Version 1.11

Tom Sgouros

May 22, 2003

Preface

This document describes version 3.2 of the Distributed Oceanographic Data System (DODS), a data system intended to allow researchers transparent access to oceanographic data—stored in any of several different file formats—across the Internet. Using DODS function libraries, many existing data analysis programs can be easily modified to accommodate access of distant datasets in a manner identical to the access of local datasets. DODS includes a protocol for the transmission of data across the Internet, and supports selection of data using constraint expressions, and translation of data from one format to another.

An overview of the system's use is presented, and specific tasks illustrated, for data providers as well as for users.

0.1 Tasks Illustrated in this Guide

For a quick start to getting, installing, and using DODS software, see the list below of tasks described in this document.

- Getting the DODS software. (page 91)
- Installing the DODS software. (page 91)
- Using a DODS client. (page 28)
- Re-linking a data analysis or display application to become a DODS client. (page 32)
- Creating and installing a DODS server.
  - Installing the DODS server and CGI filters. (page 55)
  - Starting and configuring the httpd server. Due to the variety of available servers, this task is beyond the scope of the manual. Please refer to the documentation for the particular server in question for more information.
  - Implementing a new DODS-compliant API. (*The DODS Toolkit Programmer's Guide*)
- Writing a DODS CGI program. (*The DODS Toolkit Programmer's Guide*)
- Writing the CGI service programs. (*The DODS Toolkit Programmer's Guide*)
- A list of all supported APIs. (page 30)

0.2 Who is this Guide for?

The user documentation for DODS covers two groups of users: those who want to provide access to data via DODS and those who want to use data. In many cases the people will be one and the same since most providers will also be data users.

This documentation assumes that the readers are familiar with computers, but are not necessarily programmers.

This guide also contains technical information that will be of assistance to programmers who plan to write new DODS-compliant APIs for as yet unsupported data models. Data providers and data consumers may find some general questions answered by this material, but it is not necessary to know any of it in order to use the system.

0.3 Organization of this Document

This book is organized into separate sections for data providers, data consumers, and technical reference material for programmers.

Part I is for everybody who wants to use DODS.

Chapter 1 provides a high-level overview of the entire system.

Part II is for data consumers, that is, the people who want to look at data using the DODS system.

Chapter 3 shows how to look at data using DODS. It includes a section about the theoretical and practical problems of data model translation. It also explains how to build a DODS client, which is the program used to look at DODS data.

Part III is for data providers, or people who want to make their data available through DODS servers.

Chapter 5 shows how to use DODS to make your data available to others. It explains how to set up a DODS server to provide DODS data to DODS clients, and also contains information about modifying or writing a DODS server.

Part IV contains technical information about how DODS works. This information is provided to people who want to write new libraries to use DODS through a currently unsupported API.

Chapter 6 contains general information about Data and Data Models. This is important information to have for people intending to use DODS to provide data to others. It covers the DODS data attribute and data descriptor structures. The chapter also contains a section outlining the problems associated with Data Model translation.

Appendices

Appendix A on page 91 contains the instructions for installing the DODS libraries, and software that requires these libraries.

Glossary A small but useful collection of terms.

0.4 Conventions

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

Table 1: Typographic Conventions

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

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.

Contents

List of Figures

List of Tables

Part I

Overview

1 What is DODS?

The Distributed Oceanographic Data System (DODS) provides a way for ocean researchers to access oceanographic data anywhere on the Internet from a wide variety of new *and existing* programs. By developing network versions of commonly used data access Application Program Interface (API) libraries, such as NetCDF , HDF , JGOFS , and others, the DODS project can capitalize on years of development of data analysis and display packages that use those APIs, allowing users to continue to use programs with which they are already familiar.

The DODS architecture uses a client/server model, with a *client* that sends requests for data out onto the network to some *server* , that answers with the requested data. This is exactly the model used by the World Wide Web where client programs called browsers submit requests to web servers for the data that make up web pages. Of course, DODS clients can do much more than browse this data. Using flexible data types suitable for many uses, including scientific data, the DODS servers deliver real data directly to the client program in the format needed by that client.

In fact, the network communication model used by DODS uses Uniform Resource Locator (URL) addresses and web servers (`httpd`) to deliver data to the researcher. This is done by using the DODS software to convert a researcher's data analysis software into a sophisticated (though specialized) web browser. In addition to providing network-compatible versions of popular data access APIs, the DODS project also provides a software client and server toolkit to help other developers create network-compatible DODS versions of other APIs.

To expand the universe of data available to a user, DODS incorporates a powerful data translation facility, so that data may be stored in data structures and formats defined by the data provider, but may be accessed by the user in a manner identical to the access of local data files on the user's own system. Though there are limitations on the types of data that may be translated (See Section 6.1.2 on page 69), the facility is flexible and general enough to handle many of the possible translation. There are two important results:

- A user may not need to know that data from one set are stored in a format different from data in another set. Further, it may be possible that *neither* data set is stored in a format readable by the original (i.e. without DODS) version of the data analysis and display program he or she uses.
- No segment of DODS users will be effectively cut off from accessing data because of its storage format. A scientist who wishes to make his or her data available to other DODS users may do so while keeping that data in what may actually be a highly idiosyncratic storage format. Of course, it doesn't have to be in a highly idiosyncratic format. The point is that DODS can handle a wide variety of possible cases.

The combination of the DODS network communication model and the data translation facility make DODS a powerful tool for the retrieval, sampling, and display of large distributed datasets. Though DODS was developed by oceanographers, its application is not constrained to oceanographic data. The organizing principles and algorithms may be applied to many other fi elds where data can be stored on computers.

The population of people who may be interested in a system such as DODS may be divided into data consumers and data providers. Though it was an important observation to the development of DODS that the two roles are often assumed by the same scientists, the division is a useful one for the introduction of the system. The following two sections provide a broad introduction to the roles of data consumer and data provider. The remainder of this guide is organized around this distinction between classes of users.

1.1 Why Use DODS to Read Data?

A scientist wishing to examine and sample some dataset will typically be comfortable using a relatively small number of data analysis and display programs or packages. Some of these packages will use one of the popular data access APIs currently available. However, few data access APIs provide direct access to distributed data [1], so this access must be made with network tools, such as web browsers or `ftp`. While relatively straightforward in principle, this process can nonetheless become time-consuming and somewhat challenging in practice.

The following example illustrates some of the differences between accessing distributed data with the tools currently in widespread use, and the same operation using DODS.

1.1.1 An Example: Using ftp

The advent of the WWW has made possible simple data browsers that allow sophisticated interactive sampling of on-line datasets. Using a web browser and `ftp` , a user can sample any of several large oceanographic datasets available on the Internet. However, there are several problems with these data search engines that may only become apparent when a user actually tries to use the data.

Among the problems that can arise are those that appear when a user tries to use the results of one dataset to search a second dataset. Suppose that a user wishes to choose a sea-surface temperature image from the NOAA/NASA Pathfinder AVHRR archive at:

`http://podaac-www.jpl.nasa.gov/mcsst/mcsst_subset.html`

using the results of a time-series generated from the COADS Climatology archive at:

`http://ferret.wrc.noaa.gov/fbin/climate_server`

The steps are theoretically straightforward:

❶ Create the time series from the COADS Climatology archive. This is done by answering the menu of options on the COADS web page.

❷ Import the time series from step 1 to the user's local data analysis system. Note that this step may itself require several steps:

  ❶ The data must be down-loaded, using `ftp` or a similar program.

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

    ❷ Once down-loaded, the data may have to be converted into a format that can be read by the data analysis program.

❸ Examine the data and formulate a request to the AVHRR archive. This is again done by answering the menu of option on the AVHRR Web page. Note that the COADS and AVHRR pages are not completely compatible in this respect. For example, the date formats of the two pages are different.

❹ Import the result of step 3 to the user's local data display system. This may also require several steps:

    ❶ The data must be down-loaded again.

    ❷ And again, once down-loaded, the data may have to be converted into a format that can be read by the data analysis program. Note that the set of available formats on the COADS page are distinct from the available options from the AVHRR archive.

❺ Think about the results.

Though the procedure is straightforward and the web servers designed to make sampling the datasets a simple task, upon close examination, the combination of the steps may create unforeseen difficulties. For example, a request to the COADS server will return either a spreadsheet suitable for use on a PC, a netCDF format file, or a file in one of a selection of simple ASCII formats. If the user is fortunate, the returned file will already be in a format compatible with the desired analysis package. But not all users will be so fortunate. Often this file must be converted to some other file format before it can be imported to the user's analysis program. This may or may not be a simple task.

Even a file format for which a user is properly equipped may be used in an unfamiliar manner. For example, the independent and dependent variables might be in a different order or an ASCII data file may use tabs instead of spaces.

Assuming the import of the COADS data has been accomplished and boundaries for the AVHRR search identified, the task of selecting from the second archive may begin. Unfortunately, the request to the AVHRR archive will return either a GIF picture, an HDF format file, or a raw (binary) data file. Again, importing this output into the user's analysis program may or may not be simple, but it will not be the same procedure as the one used for the first data request.

Other problems are also apparent. The COADS Climatology sampling program requests the user supply dates (month and day), whereas the AVHRR archive asks for the "Julian day" (an integer between 1 and 365 or 366). One server will accept "S" and "W" to indicate South latitudes and West longitudes, while the other requires that these be indicated with negative coordinate values. The sampling of the COADS dataset, while flexible, may not allow sampling in the manner the user needs. It cannot, for example, provide a section except along a line of

constant latitude or longitude. If a user wanted to see a section along a NE-SW line, it would be a challenging and time-consuming task to assemble one from many small data requests.

Further, it might be desirable to use the results of sampling these two databases to construct a time series. This could conceivably mean repeating the entire procedure many times.

1.1.2 An Example: Using DODS

To produce the same data selection using DODS, a user would follow essentially the same steps. However, the steps themselves would be performed differently. Once the user's data analysis package has been converted to a DODS client (Section 3.1 on page 31), the accesses to the remote datasets are made through the analysis package itself. Instead of specifying a data file by a pathname reference to some local disk file, the user specifies a URL, which may point to either a local or a remote dataset. Here is a recap of the same operation, outlined as they would be performed by a DODS application program:

❶ Create the time series from the COADS Climatology archive. This is done by using the sampling facilities of whatever data analysis program a scientist is familiar with. If desired, DODS constraint expressions may be used to reduce the network load, or to provide a sampling scheme not supported by the data analysis program.

❷ The data need not be imported to the user's data analysis program, since it was down-loaded and converted automatically in step 1.

❸ Examine the data and formulate a request to the AVHRR archive. This is again done through the sampling facilities of whatever data analysis program the user is using, and DODS constraint expressions. Note that, whatever their actual format, both COADS and AVHRR archives appear to the DODS client to be stored in identical formats.

❹ The data need not be imported to the user's data analysis program, since it was down-loaded and converted automatically in step 3.

❺ Think about the results.

It is important to note that *any* data analysis package that can handle one of the DODS-supported data access APIs can be converted into a DODS client program capable of reading data stored by *all* of the DODS-supported data access APIs. (There are some limitations on translation. See Section 1.1.3 on page 8 and Section 6.1.2 on page 69 for more information.) Therefore, assuming the user has some analysis package capable of doing the required sampling and analysis on local data, all the steps would be performed from within that package, just as if

the user were operating on local files. The result is a simpler procedure, even though the same essential steps are followed.

The DODS scenario has, among others, the following advantages:

- The user need not learn about any of the archival formats, since the DODS server and client cooperate to deliver the data in the format in which the analysis package expects to see it. Whereas the user of the ftp server has to worry about importing the data into the analysis program, the DODS client program imports it transparently.
- The user can sample the distant datasets in any fashion supported by his or her own (local) analysis package. Unnecessary data need not be sent over the Internet.
- By appending a *constraint expression* to the URLs given to the analysis program, the user can sample data using techniques that their analysis program *cannot* do.[2]
- A substantial amount of the searching and sampling is performed on the server machines. This reduces Internet traffic, as well as decreasing the load on the local machine.

1.1.3 The DODS Client

DODS uses a client/server model. As mentioned previously, the DODS servers are simply `httpd` web servers, equipped to interpret a DODS URL sent to them. (See Chapter 5.) The DODS client program can be any program that uses one of the supported APIs, such as JGOFS or netCDF.[3]

Without DODS, an application program that uses one of the common data access APIs such as netCDF will operate as shown in figure 1.1. The user makes a request for data from the application program. The program in turn uses procedures defined by the data access API to access the data, which is stored locally on the host machine. Some APIs are somewhat more sophisticated than this, of course, but their general operation is similar to this outline.

The operation of a DODS client is illustrated in figure 1.2. Here, the *same application program* that was used in figure 1.1 has been linked with a DODS version of the data access API. Now, in addition to being able to use local data as

[2]For example, suppose a user wishes to access the NODC XBT database using a program that uses the netCDF API. A program that can process the arrays that netCDF manipulates are largely unsuitable for XBT station data. However, a user can define constraint expressions in the URL to sample the data and deliver it in a form the netCDF API can use. For more information about constraint expressions, see Section 4.1. For more information about data models and translation, see Chapter 6.

[3]Or a program specially developed to read data from DODS servers.

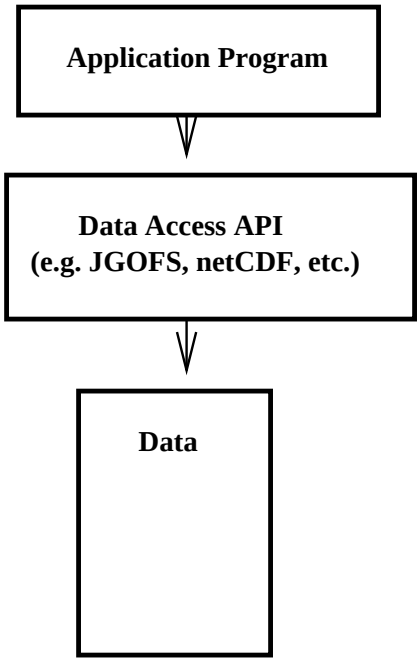

Figure 1.1: The Architecture of a Data Analysis Package.

before, the application program is able to access data from DODS server anywhere on the Internet in the same manner as the local data.

To make some program into a DODS client, it must only be re-linked with the DODS implementation of the supported API library. This is a simple process, generally requiring only a few minutes. The process will create a program that accepts URLs, specifying a location for the data somewhere on the Internet, in addition to file pathnames which only specify a location on the local platform's file system. (See Section 3.1 on page 31.)

DODS also provides a data translation facility. Data from the original data file is translated by the DODS server into a DODS data model for transmission to the client. Upon receiving the data, the client translates the data into the data model it understands. (See Chapter 6 for more information about the DODS data model.) Because the data transmitted from a DODS server to the client travel in the DODS format, the data set's original storage format is completely irrelevant to the user of a DODS client. If the client was originally designed to read netCDF format files, the data returned by the DODS-netCDF library will appear to have been read from a netCDF file, whatever the actual format of the files from which the data were read[4]. If the program expects JGOFS data, the DODS-JGOFS library will return

[4]Note that there is a limit to what can be translated. An API meant to support two-dimensional arrays may be able to handle one-dimensional vector data, but a program designed to process one-dimensional vector data will not know what to do with a two-dimensional array. The set of data access APIs supported by DODS contain several such mismatches. See Section 6.1.2 for more information.

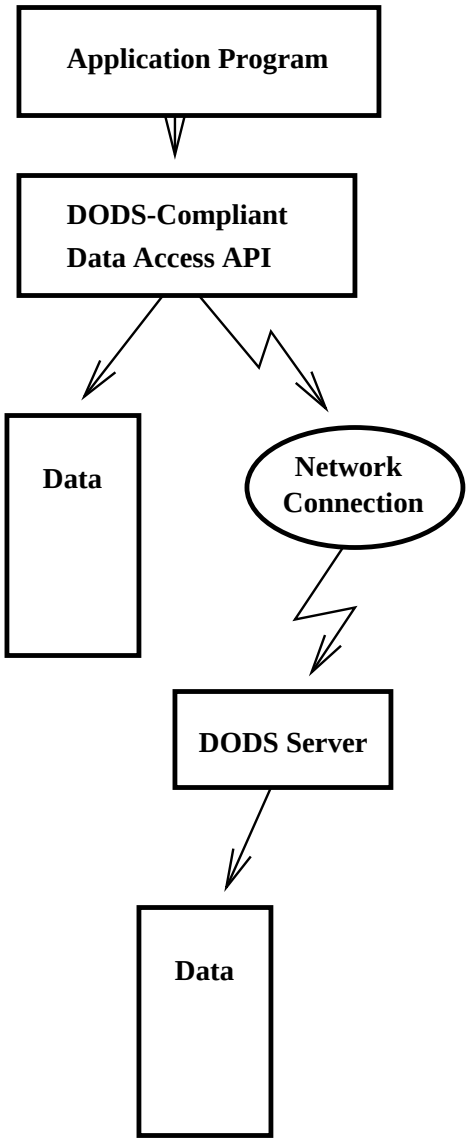

Figure 1.2: The Architecture of a Data Analysis Package Using DODS.

data that seem to have come from a JGOFS dataset, again, no matter what the actual input file format.

DODS does not pretend to remove all the overhead of data searches. A user will still have to keep track of the URLs of interesting data sets in the same way a user must now keep track of the names of files containing interesting data. A DODS *catalog service* is in the process of being constructed that will help users scan the available datasets.

1.2 Providing Data with DODS

The DODS data provider is the person or organization willing to make their digital datasets available to the community with a DODS server.

The designers of DODS recognized that many of the data users are also the data providers, and DODS was built with a recognition that providing the data should be as simple and as straightforward as possible. In many cases, once a local web server is equipped to become a DODS server, a scientist need do very little beyond what must be done simply to make the data available locally. (i.e., Put the data into a file format that can be read by the locally used data analysis and display programs.) The tasks of a data provider can be separated into three parts:

- Install and configure the DODS server. (Section 5.2 on page 55.)
- Create whatever ancillary data files are needed by the data set (if any). (Section 1.2.2 on page 12.)

1.2.1 The DODS Server

The DODS data server is simply made up of a regular `httpd` server equipped with CGI programs (or filters) that will respond to requests for dataset structure, data attributes, and data itself. (See Section 6.2 on page 71 for a description of the data returned by these requests and see Section 2.1 on page 18 for a description of the DODS URL syntax used to send these requests.) Most of the task of a data provider consists of configuring this server. While perhaps not a trivial task, it potentially represents far less effort than packaging a dataset for submission to some central data archive. Furthermore, modifying a server's configuration to accommodate new data will be an almost trivial task, involving the simple editing of a configuration file.

1.2.2 Ancillary Data

In order for a DODS client to accept data from a DODS server, it must be able to allocate the data structures and arrange internal labels to organize the incoming data. The information the client library needs to do this organizing is called the ancillary data[5]. For many APIs, the ancillary data is inherent in the data files themselves, and the DODS server can glean that information by scanning the data files. For large data archives, where scanning the data files is impractical, and that might not change often, DODS can cache the ancillary data to speed access times. When a client requests the ancillary data, the DODS server can check this data cache first before scanning the data files.

[5] It is also referred to as the Data Descriptor Structure and the Data Attribute Structure. See Chapter 6 for more details about these structures.

This feature is useful in other cases because not all data file formats are self-describing. For example, a data set might contain several files of time vs. temperature data; the header information describing which numbers are temperature and which time may be in a different file or may simply be understood by the user of the local data analysis program equipped to look at this data. As an example, data accessed by DODS servers using the FreeForm data access API require provider-created ancillary data files.

1.2.3 Administration and Centralization of Data

Under DODS, there is no central archive of data. Data under DODS is organized in a manner similar to the World Wide Web itself. That is, all one need do to make one's data available is to start up a properly configured `httpd` server on an Internet node that has access to the data to be served. Each data provider is free to join and to leave the system when it is convenient, just as any proprietor of a web page is free to delete it or add to it as whimsy demands.

Of course, as can also be seen on the World Wide Web, there are some disadvantages to the lack of central authority. If no one knows about a web site, no one will visit it. Similarly, listing a dataset in a central data catalog, such as the Global Change Master Directory (`http://gcmd.gsfc.nasa.gov/`), can make data available to other researchers in a way that simply configuring a DODS server does not. DODS provided a facility for registering a data set with the GCMD catalog, which makes the data set known to the DODS data location service.

The remainder of this book will be divided into three major sections: instructions on the building and operating of DODS clients; a tutorial and reference on running DODS servers and making data available to DODS clients; and technical documentation describing the implementation details (and the motivation behind many of the design decisions) of the DODS software.

Part II

Using DODS to Read Data

2
Using DODS

A user uses DODS with a DODS client program. This client program may have been acquired by the user (for example, the DODS Matlab and IDL graphic user interfaces, or Ferret, a freeware data analysis package each use DODS for data access), or may be a program converted to use the DODS library for data access (see Chapter 3.

In either case, there are a set of issues that must be addressed in order to use a program to access data through DODS. The issues can be classed into two groups. One set of issues involves confi guring the system to provide DODS with the helper applications and environment variables it requires. The other set concerns the manner in which a user communicates with a DODS server. We cover this fi rst.

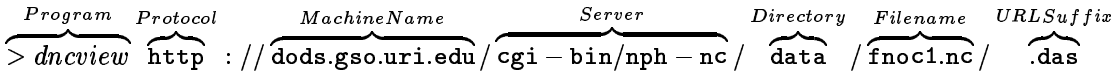

Figure 2.1: Parts of a DODS URL (without a constraint expression)

2.1 How DODS Finds Data

Once linked to the DODS libraries, a DODS client created from an existing program will work exactly as before when run using local files. However, a user can also specify a DODS Uniform Resource Locator (URL) to indicate some data file on a remote host machine. When the program receives this URL, the DODS libraries will recognize it as remote data, and issue a network request for the data. If a user has also installed a DODS server on the local machine, then local data may be accessed either through their local filenames or their DODS URL.

A URL is simply a unique name for some Internet resource. The figure 2.1 shows the parts of a typical DODS URL.

The parts of the URL are:

protocol The protocol of an Internet request may be thought of as the kind of conversation the client expects to have with the target machine. For example, a web browser like Netscape Navigator wants to find a server that can return hypertext documents, while an ftp client wants to find a server that can understand file transfer requests. A web browser equipped to display hypertext documents will specify `http` as the protocol for its conversation, and hope that the target machine has an `httpd` daemon listening.

host The host name in a URL is simply the Internet address of the host machine running whatever server can reply to the specified protocol.

server A special feature of the `httpd` server process is that it may be configured to execute Common Gateway Interface (CGI) programs upon receipt of a properly specified URL. This is used, for example, by Internet search engines that ask a user to fill out a form. The CGI specification will be specific to the server in question, and the part of the URL that follows the CGI name is passed to the CGI upon invocation. This data may include a file name, but it may as easily be some arbitrary string of instructions. The DODS server is simply a set of CGI scripts executed on demand by the `httpd` server. Here, the DODS server is represented by a CGI script called `nph-nc`.

filename If a CGI is not specified, the part of the URL after the host name is simply the name of a file that is to be returned to the inquiring browser. If a CGI is specified, the file is given to the program as its argument.

URL suffix If you are issuing a DODS request from a non-DODS client, such as a web browser, you can specify the type of request by appending a suffix to the URL. Different suffixes demand different services from the server. The different services are listed in Section 2.2 on page 20. If you are using DODS from a DODS client, or a client program adapted to use the DODS DAP library, you do not need to use a URL suffix. For example, to use DODS from Matlab, with the Matlab GUI or command-line clients, you do not need to use a suffix. To use DODS from a simple web browser like Netscape Navigator, you will need to use a suffix.

The URL in figure 2.1 shows a client request to the `httpd` server on the machine `dods.gso.uri.edu`, for a netCDF dataset (specified by the `nph-nc` in the `cgi-bin` directory) contained in a file called `fnoc1.nc`. Upon receiving this URL, the `httpd` server executes the specified DODS server module (`nph-nc`), which retrieves the file is in a directory called `data` relative to wherever the `httpd` server looks for its data[1].

DODS URLs can get somewhat more complicated than this simple description. In particular, they can contain "constraint expressions" that limit a request to data satisfying a set of conditions, and they can contain requests to specific DODS services, besides the data delivery service suggested here. Constraint expressions are described in more detail in Section 4.1 on page 36, while the array of services provided by DODS servers are described in Section 2.2 on page 20.

2.1.1 Security

Some DODS data providers will choose to control access to some or all of their data. When you request data from one of these servers, the DODS client will prompt you for a username and password. If you want to avoid the prompt, you can make the DODS URL even more baroque by embedding a username and password in it, like this:

```
http://user:password@www.dods.org/nph-dods/etc...
```

[1]The only part of the URL whose spelling is not at the discretion of the administrator of the host machine is the `http`, and the `nph-` at the beginning of the CGI script name. Even the `nc`, indicating netCDF, can be changed, although for clarity's sake, we hope people won't do so. Incidentally, the `nph-` is a relic, dating from the early days of the World Wide Web and the first hypertext protocol 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.

2.2 The DODS Services

Up to now, we have treated the DODS server as if it has only one service: providing data to clients who ask for it. It is true that this is the most important service a server provides. However, it is also true that the server provides several other services besides that. In fact, fulfilling a request for data actually requires three separate requests from the client, using three different services of the DODS server.

The services requested from a DODS server are specified in a suffix appended to the URL described in figure 2.1. Depending on the suffix supplied, the server will provide one of these services:

Data Attribute This service returns the entire data attribute structure for the given dataset. This is a text file describing the attributes of each data quantity in that dataset. (See Section 6.4.2 on page 82 for more information about data attributes.) This service is activated when the server receives a URL ending with `.das`.

Data Descriptor This service returns the entire data descriptor structure for the given dataset. This is a text file describing the structure of the variables in the dataset. (See Section 6.4.1 on page 81 for more information about data descriptors.) This service is activated when the server receives a URL ending with `.dds`.

DODS Data This service returns the actual data requested by a given URL. This is not a text file, but is encoded as a Multipurpose Internet Mail Extensions (MIME) document. This service is activated when the server receives a URL ending with `.dods`

ASCII Data This service returns an ASCII representation of the requested data. This can make the data available to a wide variety of browser programs. This service is activated when the server receives a URL ending with `.asc` or `.ascii`.

WWW Interface When the server receives a URL ending in `.html`, it produces an HTML form containing information from the dataset that you can use to construct a sensible URL with which to request DODS data. The WWW Interface is also triggered when the DODS server receives a URL that references a directory instead of a file.

Information This service returns information about the server and dataset, in human-readable HTML form. The returned document may include information about both the data server itself (e.g. server functions implemented), and the dataset referenced in the URL. The server administrator determines what information is returned in response to such a

request. This service is activated when the server receives a URL ending with `.info`. See Section 5.2.3 on page 56 for more information about how to configure the information service.

Version This service returns the version information for the DODS server software running on the server. This service is triggered by a URL ending with `.ver`.

Help This service returns some help text in response to an improperly specified URL. This service is triggered by a URL ending in any suffix that is not recognized by the DODS server.

NOTE: A request for data from a DODS client will generally make three different service requests, for data attributes, data descriptors, and for data. The prepackaged DODS clients do this for you, so you may not be aware that three requests are made for each URL. That is, a DODS client may accept a DODS URL specifying some data, such as the one shown in figure 2.1. In this case, the DODS client library (such as nc-dods) will accept the input URL, and append the different suffixes to that URL, making three distinct requests to the DODS server.

2.2.1 WWW Interface

Each DODS server implements a service called the WWW Interface. This is a way to use a standard Web client, such as Netscape, to get information about the data served by a specific server.[2] The WWW Interface has two modes of operation: the directory level and the file level.

If a DODS URL references a directory instead of a file on the server machine, the server produces a listing similar to that shown in figure 2.2.

Clicking on a dataset shown in the directory-level listing will produce an HTML form similar to the one in figure 2.3. The top line in the window ("Data URL") shows a URL that makes a request for a DODS dataset. The windows below it show the variables that make up the dataset. You can edit the form to select the data you'd like to see from this dataset, and the WWW Interface will edit the Data URL so that it only requests the data you are interested in. When done, you can push the "ASCII" button, to see an ASCII representation of the data you've requested. Netscape cannot handle binary data, so if you want to use the binary data, you should copy the URL in the Data URL window to the DODS client you'd like to use.

[2] The WWW Interface is only available for servers later than version 3.1.

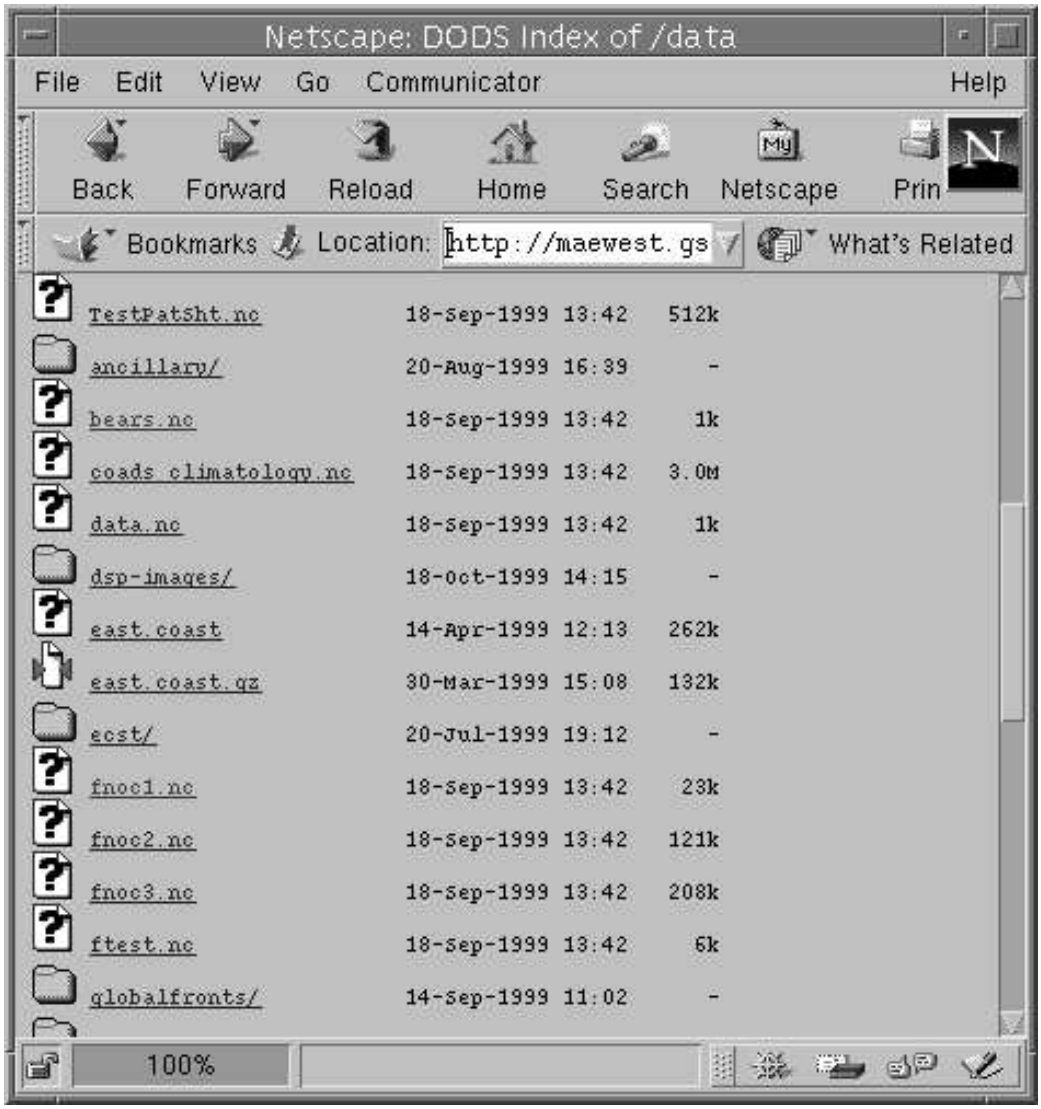

Figure 2.2: WWW Interface - Directory Level

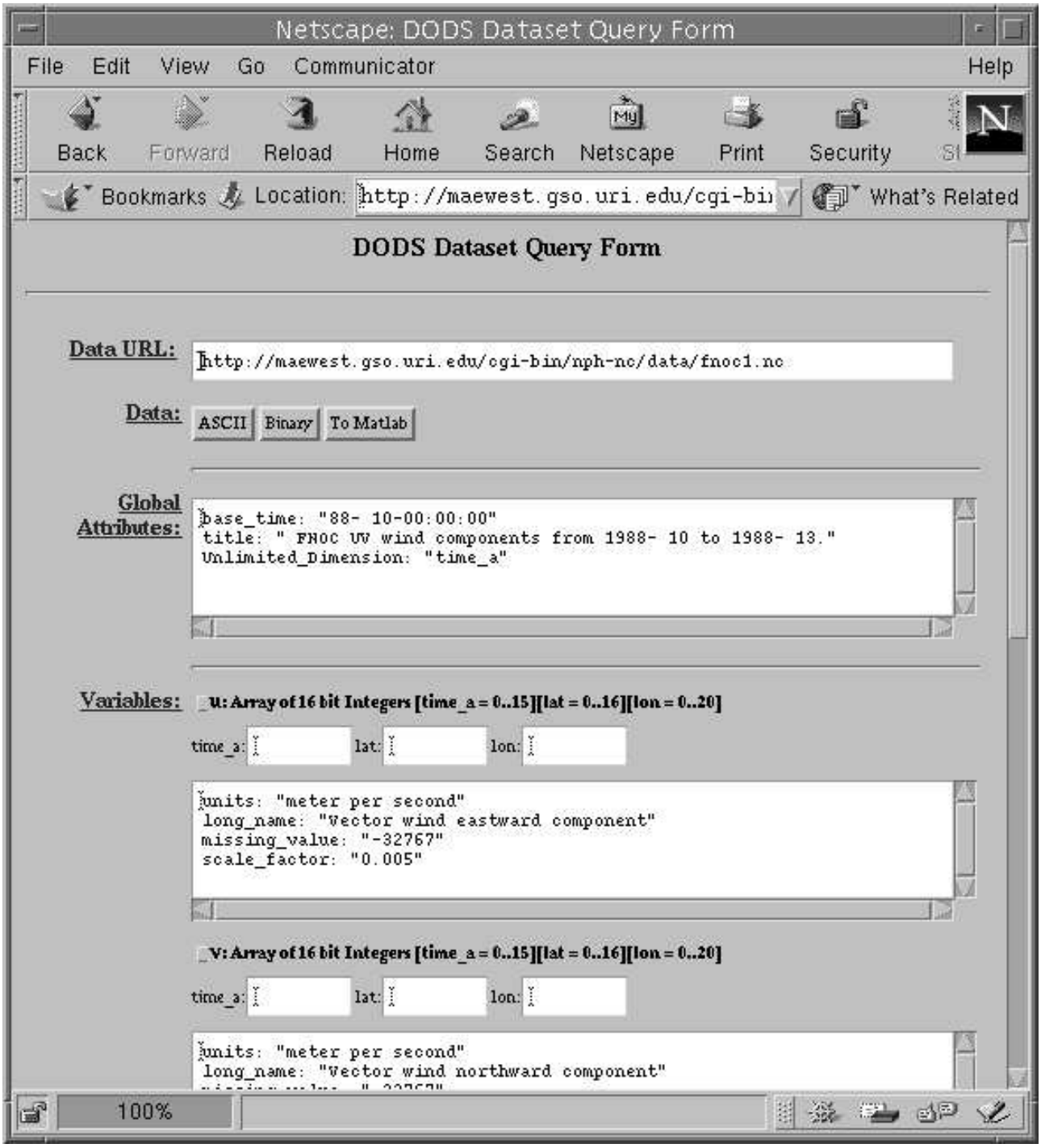

Figure 2.3: WWW Interface

2.3 Using a DODS Program

There are some configuration issues a user must consider in order to use a DODS client application program. There is a short list of software that is required for some of the advanced features of DODS, and some environment variables that control the execution of the DODS software. For a piece of software that has been converted to use DODS, after these conditions are satisfied, the program will run in the same manner it ran before. Aside from network delays, the user should not be able to tell that they are accessing data from the Internet.

Finally, though it may seem unnecessary to mention, in order for a DODS client application to communicate with a DODS server, the computer running the DODS client must be connected to the Internet.

2.3.1 Requirements

In order to use of some of the features of the DODS core software, a user's computer must have some additional software installed, and available on the user's `PATH` , in `$DODS_ROOT/bin` or `$DODS_ROOT/etc`.

- The `wish` Tcl /Tk interpreter (or whatever program is indicated by the `DODS_GUI` environment variable) is used by the *GUI manager* to provide a progress indicator that displays the status of a pending data request as it is being processed. It is also used by the error reporting system to display error message received from the server.
- The `gzip` program, the GNU compression software, is used to decompress data messages received from a DODS server. If this program is not installed, the DODS core software tells the server not to send compressed messages, so data may still be received. However, having the compression software installed and available will increase the data transfer rate.

The required software, like DODS itself, is free software. Refer to Appendix A on page 91 for information about acquiring that software.

2.3.2 Environment Variables

After successfully relinking an application program with the DODS libraries, there is a short list of environment variables that may be defined. Only `DODS_ROOT` is required. The other three variables are only used to override default values controlling the GUI manager process. Most users may safely ignore them.

`DODS_ROOT` indicates the root directory of the DODS software. The DODS core software must be able to locate utilities that are located in this directory tree.

`DODS_GUI` can contain the name of the program used by the *GUI manager* . A user might wish to change this variable to point to a "safe" Tcl/Tk interpreter; whatever program is used here must be able to process Tcl and Tk commands. The default value is the `wish` program.

`DODS_GUI_INIT` indicates the name of any initialization command required by the *GUI manager*. The default initialization string executes the Tcl program in `$DODS_ROOT/etc/dods_gui.tc1`.

`DODS_USE_GUI` may be used to turn off the GUI manager. Set the value of this variable to `no`, and the progress indicator and the error message windows will not be displayed.

NOTE: The user has substantial control over the GUI manager. You can change the program that listens for GUI commands from `wish` to anything else, and you can actually change the action of the GUI commands by editing the Tcl code in the files `dods_gui.tcl`, `error.tcl`, and `progress.tcl`. (These are in the `$DODS_ROOT/etc` directory.) However, editing these files and variables will not change the form of the messages from the DODS server, and from the core software that are meant to invoke these programs. In other words, the user may mess with these, but must be careful to leave the GUI manager in a form that will be able to process the messages it receives.

2.3.3 The Error System

The GUI manager is used to display error messages to the user. The messages themselves will vary with the server implementation. Refer to the documentation of the particular server, or consult the server's `info` Service (See Section 5.1.1 on page 51.), for a list of the error messages that might be issued by a particular server.

2.3.4 Temporary Files

Using a DODS client application will create a number of temporary files. They are created with the `tmpnam()` function, so their names will correspond to the rules for that function on your system (See the manual page for `tmpnam(3)`, or type `man tmpnam` for more information.) During normal operation, DODS will delete the temporary files it creates as it goes. However, if execution of the DODS

client is somehow interrupted, these files may remain, and will have to be deleted by hand.

3
The DODS Client

There are many different data analysis packages in use. Some packages, such as MATLAB and IDL, are commercially available, but many more are written for a specialized need or application. Many of these use one of the widely available sets of scientific data access functions (called an *Application Program Interface*, or API) such as NetCDF, JGOFS, or HDF. There is great variety among all these programs, but one feature they share is that they all access data through files containing that data[1]. That is to say that each program begins by identifying a file containing the data the user wishes to examine or analyze.

A DODS client is simply a data analysis application linked with the DODS libraries instead of the standard data access API. Using this program, a user can look at files containing data in the same way as was possible without the DODS libraries. However, by using these libraries, a user can also use a URL (URL), instead of a simple file name, to specify data located anywhere on the Internet. Figure 1.1 and figure 1.2 illustrate the operation of an application program linked with a standard data access API, and the same program linked with the DODS version of that API.

A DODS client is then a data analysis application program modified to become a web browser, somewhat like any other web browser (NCSA Mosaic, Netscape Navigator) with which you may be familiar. A web browser can only display the data it receives, however. What makes a DODS client different from another web browser is that, unlike Netscape, once the data has been received from a DODS server, the DODS client application can compute with it.

Like a web browser, a DODS client accepts a URL from a user, and parses it to come up with a protocol, an address, and a message. (See Section 2.1 on page 18 for more information about URLs.) The browser then sends a message to the address, directed to the server who can service the desired protocol, asking for the information specified in the remainder of the URL. Unlike a typical web browser,

[1]This is not true of some APIs, such as JGOFS. That API, however, uses a data dictionary to allow the user to think that the data access is through files.

a DODS client will not know what to do with data returned for a web page containing text and pictures, but a DODS server will return scientific data that a DODS client can understand and process.

Here is a simple example, using the `ncview` program. This program simply prints out the contents of a netCDF formatted data file, specified on the command line, like this:

```
> ncview fnocl.nc
```

Using DODS, this same function may be executed from any computer connected to the Internet by substituting a URL for the filename above:

```
> dncview http://dods.gso.uri.edu/cgi-bin/nc/data/fnocl.nc
```

(See figure 2.1 Aside from the fact that the data is remote, and must be specified with a URL, the program will seem to function in the same way it had with the simple netCDF library (albeit somewhat more slowly due to having to make network connections instead of local file operations). You can find `dncview` (the `ncview` program linked with the DODS library) in the

```
$DODS_ROOT/src/nc-dods/ncview
```

directory. Running the above command will produce the following output:

```
netcdf fnocl {
dimensions:
    time_a = 16
    lat = 17 ;
    lon = 21 ;
    time = 16 ;

variables:
    long u(time_a, lat, ion) ;
        u:units = ‘‘meter per second’’ ;
        u:long_name = ‘‘Vector wind eastward component’’ ;
        u:missing_value = ‘‘-32767’’ ;
        u:scale_factor = ‘‘0.005’’ ;
    long v(time_a, lat, ion) ;
        v:units = ‘‘meter per second’’ ;
        v:long_name = ‘‘Vector wind northward component’’ ;
        v:missing_value = ‘‘-32767’’ ;
        v:scale_factor = ‘‘0.005’’ ;
    double lat(lat) ;
        lat:units = ‘‘degree North’’ ;
    double lon(lon) ;
        lon:units = ‘‘degree East’’ ;
    double time(time) ;
        time:units = ‘‘hours from base_time’’ ;

// global attributes:
        :base_time = ‘‘88- 10-00:00:00’’ ;
        :title = ‘‘FNOC UV wind components
                          from 1988- 10 to 1988- 13.’’ ;
data:
 u =
  -1728, -2449, -3099, -3585, -3254, -2406, -1252,
    662, 2483, 2910, 2819, 2946, 2745, 2734,
  2931, 2601, 2139, 1845, 1754, 1897, 1854, -1686,
...
```

Although there are packaged DODS browsing programs that a user can use to look at data, the user can also construct his or her own. Linking a DODS API with an already existing program allows a user to create a customized web browser that can access data available from any DODS server connected to the Internet.

The DODS APIs are designed to accurately mimic the behavior of several different commonly used scientifi c data APIs. As of this writing (May 22, 2003), the DODS API set includes:

Table 3.1: Supported APIs

API	Description	Components
netCDF	Support for gridded data, such as satellite data, interpolated ship station data, or current meter data.	Server and client.
JGOFS	Support for relational data, such as *Sequences*. Created by the Joint Globar Ocean Flux Study (JGOFS) project for use with oceanographic station data.	Server and client.
HDF	Support for gridded data. Commonly used for astronomical data and model data.	Server only.
DSP	Oceanographic and geophysical satellite data. Provides support for image processing. Developed at the University of Miami/RSMAS. Primarily used for AVHRR and CZCS data.	Server only.
GRIB	Support for gridded binary data. GRIB is the World Meteorological Organization (WMO) format for the storage of weather information and the exchange of weather product messages.	Server only, due in early 1999.
BUFR	The WMO's standard set of codes for the transmission and storage of meteorological data, using a compressed format with each data value occupying the least number of bits necessary to contain its range of values. Suitable for meteorological observations made from a single point or set of points.	Server only, due in early 1999.
Free-Form	On-the-fly conversion of arbitrarily formatted data, including relational data and gridded data. May be used for sequence data, satellite data, model data, or any other data format that can be described in the flexible FreeForm format definition language. This server can be used to serve data stored in almost all home-grown data formats.	Server only; no client required.
native DODS	The DODS class library may be used directly by a client program. It supports relational data, array data, gridded data, and a flexible assortment of data types that can be combined to c accommodate most data models.	Client.

The API set is extensible, meaning that developers can use the DODS software toolkit to write DODS-compliant versions of new APIs. See *The DODS Toolkit Programmer's Guide* for more information.

The most important result of this architecture is that, just as the use of the

`dncview` program above is identical to the original `ncview`, a user can use remote DODS data *and* continue to use the same data analysis and display programs with which he or she is familiar. Any program that uses one of the DODS-supported APIs may be re-linked to use the DODS version of that API. This creates a DODS client. That and a connection to the Internet, are all that a researcher requires to gain access to the available DODS data.

3.1 Configuring Programs to Use DODS

Relinking an existing program with the DODS implementation of some data API is a simple procedure. Find the directory that contains the source/object code of the program you want to re-link and modify the makefile (typically called `Makefile`) for the program so that the DODS-compliant API library is used in place of the standard API library. (If you can't find the libraries on your system, see Appendix A on page 91, or ask the system administrator.) These libraries are:

`libdap++.a` Software common to all of the DODS-supported APIs.

DODS also uses facilities from some standard libraries, and these must also be included in the link to resolve all the symbols.

`libwww.a` The World Wide Web library. This contains the functions used to communicate between the DODS client and server.

`libexpect.a` Functions from the `expect` library are used to communicate between DODS client processes.

`libtcl.a` Contains definitions necessary for the `expect` library. The use of this library in the link is not related to the use of Tcl by DODS clients.

`libstdc++.a` The GNU C++ class library (This is not necessary if using `g++` to re-link.)

You will also need to include the library containing the DODS-compliant version of the API. The name of this library of course depends on the API, but it is generally in the form

`lib` *API*`-dods.a`

Where *API* is an abbreviation indicating the API emulated by the specified library. For example, the DODS-compliant netCDF library is called `libnc-dods.a` and the JGOFS version is `libjg-dods.a`.

3.1.1 An Example Using netCDF

The `ncview` program is a simple utility that prints the contents of a netCDF-format file to standard output. This section outlines the process used to modify the `ncview` makefile to link that program with the DODS netCDF API, thereby turning `ncview` into a network-ready DODS client. The process of linking any other program with the corresponding DODS library is entirely analogous to this one and only requires the substitution of the program name and the appropriate library.

First the link flags were modified so that the library search path would include the likely places to find the DODS libraries:

```
LDFLAGS = -g -L$(DODS_ROOT)/lib
```

`DODS_ROOT` is an environment variable that indicates the root directory of the DODS installation, and in this manual is used as shorthand for this directory. It is typically called something like `/usr/local/DODS`. If you cannot find these directories on your system, consult your system administrator, or refer to Appendix A on page 91 for information about acquiring and installing the DODS software.

After the link flags were modified, the DODS libraries were added to the list of libraries used. The order in which the libraries are listed is important.

```
LIBS = -lnc-dods -ldap++ -lnc-dods -ldap++ -lwww -ltcl
       -lexpect -lz -lrx
```

> **NOTE:** Because DODS is implemented as a core set of classes contained in one library (`libdap++.a`) and a set of specializations of those classes in a second library (`libnc-dods.a`), and because there is a circular dependence between those two libraries, they must be included twice in the linker command.

Finally, `g++` was substituted for the link command.[2]

3.1.2 Potential Problems

When a user links an existing a program to the DODS libraries, there are several possible conditions that may cause problems.

- Some programs use more than one API.
- Some programs access data using both API and UNIX system calls.
- Some programs use undocumented features of the APIs.

[2] It is possible to use `gcc` instead of `g++`, but in that case, `-lg++` must be added to the end of the library list.

If this is the case for a given program, there is generally no good solution beside rewriting the software to conform to a strict usage of the data reading parts of the given API. Of course if the problem is that the program uses more than one API, you can try linking the program with a DODS-compliant version of the second API as well.

➨ Re-linked programs can be very large.

The DODS libraries are large, and the `g++`, `www`, `expect`, and `tcl` libraries on which they are built are even larger. This means that the executable version of a re-linked DODS client can seem unreasonably obese. Much of the disk space is occupied by symbol tables, which can be removed from the executable file with the `strip` utility. In many cases, a user can recover a substantial amount of disk space this way.

CAUTION: Without familiarity with the DODS software, it is best only to strip the executable files. Stripping object files or libraries might leave them in a useless condition for the linker. Furthermore, stripping an executable file removes symbol names, which may make diagnosing problems more difficult.

The DODS libraries only affect the data *reading* functionality of the specified API. There are no DODS replacements for functions like netCDF's `ncputrec()`, that *write* data to a disk file. These functions are included in the DODS-compliant API library, but they operate in a manner identical to the original (non-DODS) versions, that is, they work on local files only, attempting to write "over the network" will result in an error.

3.2 Writing New DODS Programs

The DODS software may also be used to write new programs. This may be done either through one of the DODS-supported API libraries, such as netCDF or JGOFS, or by using the DODS data access protocol directly. There are advantages and disadvantages to each approach.

The biggest advantage of writing new code using a DODS-supported API such as netCDF or JGOFS is that the programmer in question is probably already familiar with the use of that API. Writing a DODS program using an adapted API is not signifi cantly different than writing the same program with the original API. While writing this new program, it will be useful to remember that the data the program uses will often be remote, implying that data retrieval may not be instantaneous, and that implementation of local caching to store requested data might be a good idea, but other than that, the process is the same as writing a program using the regular API.

It is also possible to use the DODS data access protocol directly. This is somewhat more involved than using one of the DODS-compliant API libraries, and C++ is the only language supported for this. However, this approach can provide substantially more effi cient programs. For further information about this approach, refer to the technical information about the DAP in *The DODS Toolkit Programmer's Guide*.

4
Data Analysis with DODS

The DODS software is not only a data transport mechanism. Using DODS, you can subsample the data you are looking at. That is, you can request an entire data file, or just a small piece of it.

4.1 Selecting Data: Using Constraint Expressions

The URL such as this one:

```
http://dods.gso.uri.edu/cgi-bin/nph-nc/data/buoys.nc
```

refers to the entire dataset contained in the buoys.nc file. A user may, however, choose to sample the dataset simply by modifying the submitted URL. The *constraint expression* attached to the URL directs that the data set specified by the first part of the URL be sampled to select only the data of interest from a dataset even for programs that do not have a built-in way to accomplish such selections. This can vastly reduce the amount of data a program needs to process, and reduce the network load of transmitting that data to the client.

4.1.1 Constraint Expression Syntax

A constraint expression is appended to the target URL following a question mark, as in the following examples:

```
http://oceans.univ.edu/cgi-bin/nc/expl/buoys.nc?temp

http://oceans.univ.edu/cgi-bin/nc/expl/buoys.nc?temp[1,100,5]

http://oceans.univ.edu/cgi-bin/nc/expl/buoys.nc?u&lat>15.0

http://oceans.univ.edu/cgi-bin/nc/expl/buoys.nc?cast.02<15.0

http://oceans.univ.edu/cgi-bin/nc/expl/buoys.nc
                                    ?station&station.temp<15.0
```

A constraint expression consists of two parts: a *projection* and a *selection*, separated by an ampersand (`&`). Either part may contain several sub-expressions. Either part may be present, or both.

$$proj\_1, proj\_2, \ldots, proj\_n \& sel\_1 \& sel\_2 \& \ldots \& sel\_m$$

A projection is simply a comma-separated list of the variables that are to be returned to the client. If an array is to be subsampled, the projection specifies the manner in which the sampling is to be done. If the selection is omitted, all the variables in the projection list are returned. If the projection is omitted, the entire dataset is returned, subject to the evaluation of the selection expression. The projection can also include functional expressions of the form:

$$function(arg\_1, arg\_2, \ldots, arg\_n)$$

where the arguments are variables from the dataset, scalar values, or other functions.

A simple selection expression is a boolean expression of the form

variable operator variable
or
variable operator value
or
function($arg\_1, arg\_2, \ldots, arg\_n$*)*

Where

operator can be one of the relational operators listed in table 4.1 on page 40;

variable can be any variable recorded in the dataset;

value can be any scalar, string, function, or list of numbers (Lists are denoted by comma-separated items enclosed in curly braces ,for example, {3,11,4.5}.); and

function is a function defined by the server to operate on variables or values, and to return a boolean value (See Section 4.1.3 on page 41).

Each selection clause begins with an ampersand (`&`) representing the "AND" boolean operation[1].

> **NOTE:** The `&` is actually a *prefix* operator, not an infix operator. That is, it must appear at the beginning of each selection clause, no matter what. This means that a constraint expression that contains no projection clause must still have an `&` in front of the first selection clause.

There is no limit on the number of selection clauses that can be combined to create a compound constraint expression. Data that produces a true (non-zero) value for the entire selection expression will be included in the data returned to the client by the server. If only a part of some data structure, such as a *Sequence*, satisfies the selection criteria, then only that part will be returned.

> **NOTE:** Due to the differences in data model paradigms, selection is not implemented for the DODS array data types, such as *Grid* or *Array*. However, many DODS servers implement selection functions you can use for the same effect. You can query the server for the functions it implements with the usage service outlined in Section 4.1.3 on page 41.

[1] The "OR" function may be implemented with a list. For example, to say that *i* must equal 3 OR 11 you would write *i* = {3,11}The clause evaluates to true when *i* equals any one of the elements.

Simple Constraint Expression Examples

Consider the data descriptor in figure 4.1.1. The figure is an example of the Data Descriptor Structure , one of the messages returned by a DODS server in response to a query about some dataset. The full syntax description for this structure is given in Section 6.4 on page 81. For the moment, it is only important that it is the description of a dataset containing station data including temperature, oxygen, and salinity. Each station also contains 20 oxygen data points, taken at 20 fixed depths, used for calibration of the data.

The following URL will return only the pressure and temperature pairs of this dataset. (Note that the constraint expression parser removes all spaces, tabs, and newline characters before the expression is parsed.) There is only a projection clause, without a selection, in this constraint expression[2].

```
Dataset {
   Sequence{
      Int32 day;
      Int32 month;
      Int32 year;
      Float64 lat;
      Float64 lon;
      Float64 O2cal[20];
      Sequence{
         Float64 press;
         Float64 temp;
         Float64 O2;
         Float64 salt;
      } cast;
      String comments;
   } station;
} arabian-sea;
```

Figure 4.1: Sample Data Descriptor

```
http://oceans.edu/cgi/nph-jg/exp1O2/cruise?station.cast.press,
                                                station.cast.temp
```

Incidentally, we have assumed that the dataset was stored in the JGOFS format[3] on the remote host `oceans.edu`, in a file called `exp1O2/cruise`. For the sake of brevity, from here on we will omit the first part of the URL, to concentrate on the constraint expression alone.

[2] For the sake of clarity, this and several of the following constraint expression examples span multiple lines. While the constraint expression evaluator ignores newline characters, program limitations of the DODS client will likely prevent a user from typing a newline in a constraint expression.

[3] Because it contains an array, the dataset pictured in Figure 4.1.1 is technically not a valid JGOFS dataset. We have included the array for pedagogical purposes, and hope that the JGOFS purists will forgive us.

If we only want to see pressure and temperature pairs below 500 meters deep, we can modify the constraint expression by adding a selection clause.

```
?station.cast.press,station.cast.temp&station.cast.press>500.0
```

In order to retrieve all of each cast that has any temperature reading greater than 22 degrees, use the following:

```
?station.cast&station.cast.temp>22.0
```

Simple constraint expressions may be combined into compound expressions with logical AND (`&`). To retrieve all stations west of 60 degrees West and north of the equator:

```
?station&station.lat>0.0&station.lon<-60.0
```

As was mentioned, the logical OR can be implemented using a list of scalars. The following expression will select only stations taken north of the equator in April, May, June, or July.

```
?station&station.lat>0.0&station.month={4,5,6,7}
```

If our dataset contained a field called `monsoon-month`, indicating the month in which monsoons happened that year, we could modify the last example search to include those months as follows:

```
?station&station.lat>0.0
        &station.month={4,5,6,7,station.monsoon-month}
```

In other words, a list can contain both values and other variables. If monsoon-month was itself a list of months, a search could be written as:

```
?station&station.lat>0.0&station.month=station.monsoon-month
```

For arrays and grids, there is a special way to select data within the projection clause. Suppose we want to see only the first five oxygen calibration points for each station. The constraint expression for this would be:

```
?station.02cal[0:4]
```

By specifying a *stride* value, we can also select a *hyperslab* of the oxygen calibration array:

```
?station.02cal[0:5:19]
```

This expression will return every fifth member of the `02cal` array. In other words, the result will be a four-element array containing only the first, sixth, eleventh, and sixteenth members of the `02cal` array. Each dimension of a multi-dimensional arrays may be subsampled in an analogous way. The return value is an array of the same number of dimensions as the sampled array, with each dimension size equal to the number of elements selected from it.

4.1.2 Operators, Special Functions, and Data Types

The data types accessible through the DODS software are listed and described in Section 6.3 on page 72. It is advisable to be familiar with these types before trying to construct complex constraint expressions.

The constraint expression syntax defines a number of operators for each data type. These operators are listed in table 4.1

Except for the $*$ operation defined on the URL data type, all the operators defined for the scalar base types are boolean operators whose result depends on the specified comparison between its arguments. Refer to Section 4.1.4 on page 41 for a description of the URL data type and its operator.

The $\sim=$ operator returns true when the character string on the left of the operator matches the regular expression on the right. See Section 4.1.5 on page 42 for a discussion of regular expressions.

The *Structure*, *Sequence*, and *Grid* data types are each composed of a collection of simpler data types. The . and operators allow a user to refer to the subsidiary variables within these compound types. For example, `station.year` indicates the value of the `year` member of the `station` sequence.

The array operator [] is used to subsample the given array. See page 39 for an explanation and example of its use.

Table 4.1: Constraint Expression Operators.

Class	**Operators**
Simple Types	
Byte, Int32, UInt32, Float64	`< > = != <= >=`
String	`= != ~=`
URL	`*`
Compound Types	
Array	`[start:stop]` `[start:stride:stop]`

List	`length(`*list*`),`
	`nth(`*list*`,`*n*`),`
	`member(`*list*`,`*elem*`)`

Structure	`.`
Sequence	`.`
Grid	`[start:stop]` `[start:stride:stop] .`

There are three special functions defined to operate on the *List* data type. The

`length()` function returns the number of elements in the given list, the `nth()` function returns the list element indicated by the input index, and the `member()` function, which returns true if the given value equals any member of the list. Note that the behavior of the `nth()` function is undefined for indices beyond the range of the list.

4.1.3 Using Functions in a Constraint Expression

A DODS data server may define its own set of functions that may be used in a constraint expression. For example, the data server containing the example data from figure 4.1.1 might define a `sigma1()` function to return the density of the water at the given temperature, salinity and pressure. A query like the following would return all the stations containing water samples whose density exceeded $1.0275 g/cm^3$.

```
?station.cast&sigma1(station.cast.temp,
                     station.cast.salt,
                     station.cast.press)>27.5
```

Functions like this one are not a standard part of the DODS architecture, and may vary from one server to another. A user may query a server for a list of such functions by sending a URL ending with ".info". For example, you can query the data server installed on the DODS home site with the following URL:

`http://dods.gso.uri.edu/cgi-bin/nph-nc/fnoc1.nc.info`

The data returned will be an HTML message, readable with a standard web browser, containing documentation of the server running on the given site, and the data named in the URL. In this case, you will learn that the specified server defines two functions that can be used in a constraint expression:

geolocate(*variable, lat1, lat2, lon1, lon2*) Returns the elements of *variable* that fall within the box created by (*lat1*,*lon1*) and (*lat2*,*lon2*).

time(*variable, start_time, stop_time*) Returns the elements of *variable* that fall within the time interval *start_time* and *stop_time*.

4.1.4 Using URLs in a Constraint Expression

The DODS data access protocol defines a special data type to handle distributed data: *URL*. This is a scalar data type, much like the *String* type, intended to hold one DODS URL. It generally points at some remote dataset or data value. Using this data type, a constraint expression may make the data returned from one DODS data server dependent on data held at an entirely different site.

In order to accommodate this data type, DODS defines a special "dereference" operator `*`. Similar to its function with pointers in C, applying this operator to a

URL returns the data specified by that URL. The *URL* data type itself contains only a character string. It must be dereferenced to produce a reference to the data named by the URL.

Examples

The following example will return all the stations containing oxygen values greater than fifteen:

```
?station&station.cast.O2>15.0
```

Similarly, the following constraint expression will yield all the stations in the dataset whose value is greater than that of the oxygen value indicated by the URL:

```
?station&station.cast.O2>*''http://ocean.edu/etc/nc/data?O2MAX''
```

Finally, suppose that the dataset itself contained a variable of type *URL*, and that this URL contained the address of oxygen data stored at some other site. The data descriptor for the dataset might look like the following:

```
Dataset {
   Sequence{
        .
        .
        .
      URL O2cal;
        .
        .
        .
   } station;
} arabian-sea;
```

We can now write the previous constraint as:

```
?station&station.cast.O2>*O2cal
```

URLs stored in remote datasets may also be used in the projection clause of the constraint expression. Imagine a dataset that consists only of a list of URLs for each square degree of latitude and longitude. A user could query this dataset for the actual list of URLs, or, by using the `*` operator, could construct a constraint expression that would return the actual data indicated by the URLs in the target dataset.

4.1.5 Pattern Matching with Constraint Expressions

There are three operators defined to compare one *String* data type to another. The `=` operator returns TRUE if its two input character strings are identical, and the `!=` operator returns TRUE if the *Strings* do not match. A third operator, $\sim=$ is

provided that returns TRUE if the *String* to the left of the operator matches the regular expression in the *String* on the right.

A regular expression is simply a character string containing wildcard characters that allow it to match patterns within a longer string. For example, the following constraint expression might return all the stations on the sample cruise at which a shark was sighted:

```
?station&station.comment~=‘‘.*shark.*’’
```

Most characters in a regular expression match themselves. That is, an “f” in a regular expression matches an ”f” in the target string. There are several special characters, however, that provide more sophisticated pattern-matching capabilities.

`.`

The period matches any single character except a newline.

`* + ?`

These are postfix operators, which indicate to try to match the preceding regular expression repetitively (as many times as possible). Thus, `o*` matches any number of `o`’s. The operators differ in that `o*` also matches zero `o`’s, `o+` matches only a series of one or more `o`’s, and `o?` matches only zero or one `o`.

‘[...]’

Define a “character set,” which begins with `[` and is terminated by `]`. In the simplest case, the characters between the two brackets are what this set can match. The expression `[Ss]` matches either an upper or lower case `s`. Brackets can also contain character ranges, so `[0-9]` matches all the numerals. If the first character within the brackets is a caret (`^`), the expression will only match characters that do not appear in the brackets. For example, `[^0-9]*` only matches character strings that contain no numerals.

`^ $`

These are special characters that match the empty string at the beginning or end of a line.

`\|`

These two characters define a logical OR between the largest possible expression on either side of the operator. So, for example, the string `Endeavor\|Oceanus` matches either `Endeavor` or `Oceanus`. The scope of the OR can be contained with the grouping operators, `\(` and `\)`.

`\( \)`

These are used to group a series of characters into an expression, or for the OR function. So, for example, `\(abc\)*` matches zero or more repetitions of the string `abc2`.

There are several more special characters and several other features of the characters described here, but they are beyond the scope of this guide. The DODS regular expression syntax is the same as that used in the Emacs editor. See the documentation for Emacs [1] for a complete description of all the pattern-matching capabilities of regular expressions.

Examples

In the above example, a user might wonder whether the shark comments had been spelled with upper or lower case letters. The following constraint expression will return any station that mentions a shark in upper or lower case.

```
?station&station.comment~=‘‘.*\(SHARK\|shark\).*’’
```

Of course, this would miss `Shark` and `sHark` and so on. The constraint could be written this way to catch all odd permutations of upper and lower case:

```
?station&station.comment~=‘‘.*[Ss][Hh][Aa][Rr][Kk].*’’
```

4.1.6 Optimizing the Query

Using the tools provided by DODS, a user can build quite elaborate and sophisticated constraint expressions that will return precisely the data he or she wishes to examine. However, as the complexity of the constraint expression increases, so does the time necessary to process that expression. There are some techniques a user may user to optimize the evaluation of a constraint that will ease the load on the server, and provide faster replies to DODS dataset queries.

The DODS constraint expression evaluator uses a ”lazy evaluation ” algorithm. This means that the sub-clauses of the selection clause are evaluated in order, and parsing halts when any sub-clause returns FALSE. Consider a constraint expression that looks like this:

```
?station&station.cast.O2>15.0&station.cast.temp>22.0
```

If the server encounters a station with no oxygen values over 15.0, it does not bother to look at the temperature records at all. The first sub- clause evaluates FALSE, so the second clause is never even parsed.

A careful user may use this feature to his or her advantage. In the above example, the order of the clauses does not really matter; there are the same number of temperature and oxygen measurements at each station. However, consider the following expression:

```
?station&station.cast.O2>15.0&station.month={3,4,5}
```

For each station there is only one month value, while there are many oxygen values. Passing a constraint expression like this one will force the server to sort through all the oxygen data for each station (which could be in the thousands of

points), only to throw the data away when it finds that the month requested does not match the month value stored in the station data. This would be far better done with the clauses reversed:

```
?station&station.month={3,4,5}&station.cast.O2>15.0
```

This expression will evaluate much more quickly because unwanted stations may be quickly discarded by the first sub-clause of the selection. The server will only examine each oxygen value in the station if it already knows that the station might be worth keeping.

This sort of optimization becomes even more important when one of the clauses contains a URL. In general, any selection sub-clause containing a URL should be left to the end of the selection. This way, the DODS server will only be forced to go to the network for data if absolutely necessary to evaluate the constraint expression.

4.2 A Word About Data Translation

Once a researcher is freed from the confines of using only local data, he or she will soon discover that there is a wealth of data available on the Internet, and nearly all of it is stored in formats incompatible with her own. Worse, the data formats are often mutually incompatible, rendering the confusion complete. DODS provides a solution applicable to a great many such problems.

When a DODS server retrieves data from some distant machine, that data may be in any of several file formats supported by DODS. The server translates the data, however, into an intermediate format for transmission. Upon receipt of the messages containing data, the DODS client software unpacks the data into the form expected by the calling client program and returns it to that program. Because all data must be translated into the same intermediate format, DODS becomes a powerful format translator for datasets. In effect, this means that a program designed to read and display JGOFS data can look at the DODS data catalog and see everything as JGOFS datasets. A netCDF program can look at those same datasets, from that same catalog, and think they are all in netCDF format. This system of translation allows a researcher to ignore the question of formats and concentrate on the data alone.

Of course, there are some translations that cannot be done transparently, if they can be done at all. Consider a two-dimensional array of satellite sea-surface temperature measurements. Assume the data is stored in netCDF format on some machine called `satt.uri.edu`. The data might be uniquely specified by some URL, say `http://satt.uri.edu/sst/010694.nc`. However, were a user to feed that URL to a JGOFS-originated DODS client designed to draw property vs. depth graphs of station data, no translation facility would be able to map the original data into a form accommodated by the client program.

The issues of data models and data translation are important ones to the data provider. These issues are discussed in detail in Section 6.1.2 on page 69

Part III

Providing Data with DODS

5
The DODS Server

See the DODS Server installation guide for most of this information.

There are two separate pieces to the task of installing a DODS data server: installing and configuring the server itself, and telling the universe of possible users about it.[1] Only the first will be considered in this chapter. DODS provides avenues for doing the second, including a Catalog Service indexing DODS datasets, and cooperation with the Global Change Master Directory, but these are still under construction.

A DODS server is nothing more than a World Wide Web server (`httpd`) equipped with Common Gateway Interface (CGI) programs that enable it to respond to requests for data from DODS client programs. Web servers and CGI programs are standard parts of the Web, and the details of their operation and installation are beyond the scope of this guide. For further information about these, consult one of the many World Wide Web references now available. For the purposes of understanding the DODS architecture, a user need only understand the following:

- A Web server is a process that runs on a computer (the host machine) connected to the Internet. When it receives a URL from some Web client, such as a user somewhere operating Netscape or Mosaic, it packages and returns the data specified by the URL to that client. The data can be text, as in a web page, but it may also be images, sounds, a program to be executed on the client machine, or some other data.
- A properly specified URL can cause a Web server to invoke a CGI program on its host machine, accepting the input that would have gone to the httpd, and returning the output of that program to the client who sent the URL in the first place. The CGI is executed on the server. The DODS server relies on this facility.

[1] The second step is, of course, optional.

5.1 Server Architecture

A request for data made to the client DODS library will result in three different requests for data to a DODS server. The user simply enters a single URL, as described in Section 2.1 on page 18. The core DODS software then modifies the URL into three slightly different forms, and makes three requests for data to the server. The first request is for the "shape" of the data, and consists of the dataset descriptor structure, described on page 81. The second request is for the attributes of the data types described in the DDS. This structure is described on page 82. The last request is actually for the data.

The response to the DDS and DAS request URLs is text formatted using the grammars in table 6.4 and table 6.3. This text can then be parsed by the caller to determine the structure of the dataset, types and sizes of each of its components and their attributes. Depending on the data access API used to access the data, these structures may be derived either from information contained in the dataset or from ancillary information supplied by the dataset maintainers in separate text files, or both. The data in these structures (which can be thought of as data about the real data) may be cached by the client system.

The DODS DAP is a stateless protocol. The protocol *entry points* may be thought of as the different messages to which a DODS server will respond. (A message is just a URL specifying a request.) Each of the protocol entry points does a single isolated job and they can be issued in any order. Of course, it may not make sense to the user to ask for the data before asking for the data description structure, but that is not the server's problem. This separability allows the user to cache data locally if need be, so that future accesses to the same dataset can skip the retrieval of these structures.

To understand the operation of the DODS server, it is useful to follow the actions taken to reply to a data request. The diagram in figure 5.1 lays out the relationship between the various entities. Consider a DODS URL such as the following:

```
http://dods.gso.uri.edu/cgi-bin/nph-nc/data/fnoc43.nc
```

When this URL is submitted to a DODS client, it will contact the Web server (httpd) running on the platform, `dods.gso.uri.edu`. When the connection has been established, the client will forward to the server the remaining parts of the URL: `/cgi-bin/nph-nc/data/fnoc43.nc`. As the server parses this string, it will notice that `cgi-bin` corresponds to the name of the directory where it keeps its CGI programs. (The actual directory name is specific to the particular web server used, and the details of its installation. Typically, the web server documnetation might call it the `ScriptAlias` directory, and it might refer to something like `/usr/local/etc/httpd/cgi-bin`.) It looks in that directory to see whether there exists a CGI program called `nph-nc`, which is the name of the netCDF DODS server packaged with DODS. Finally, the server executes that program, specifying the rest of the URL (`data/fnoc43.nc` in this case) for an

argument. The standard output of the CGI program is redirected to the output of the `httpd`, so the client will receive the program output as the reply to its request.

For APIs that are designed to read and write files, such as netCDF, the CGI program will be executed with the working directory specified by the `httpd` configuration. On the `dods.gso.uri.edu` server, for example, all CGI programs are executed native to the directory `/usr/local/spool/http`. The last section of the URL, then, specifies the file `fnoc43.nc` in the directory:

```
/usr/local/spool/http/data.
```

Several existing data APIs, such as JGOFS, are not designed with file access as their fundamental paradigm. The JGOFS system, for example, uses an arrangement of "dictionaries" that define the location and method of access for specified data "objects." A URL addressing a JGOFS object may appear to represent a file, like the netCDF URL above.

```
http://dods.gso.uri.edu/cgi-bin/nph-jg/station43
```

However, the identifier (`station43`) after the CGI program name (`nph-jg`) represents, not a file, but an entry in the JGOFS data dictionary. The entry will, in turn, identify a file or a database index entry (possibly on yet another system) and a method to access the data indicated. (The `httpd` server must be a valid JGOFS user to have access to the dictionary.)

Note that the name and location of the `cgi-bin` directory, as well as the name and location of the working directory used by the CGI programs, are local configuration details of the particular web server in use. The location of the JGOFS data dictionary is a configuration issue of the JGOFS installation. That is to say these details will probably be different on different machines.

5.1.1 Service Programs

At this point, the request for data, encoded in a URL, has caused the `httpd` server to execute the CGI program that represents the DODS server. The DODS server, in turn, executes one of several different service programs, and returns the result of that execution to the client. Though there may be others available on a given machine, five of the services constitute the core functionality of the DODS server:

- Data Attribute
- Data Description
- Data
- ASCII Data
- Information

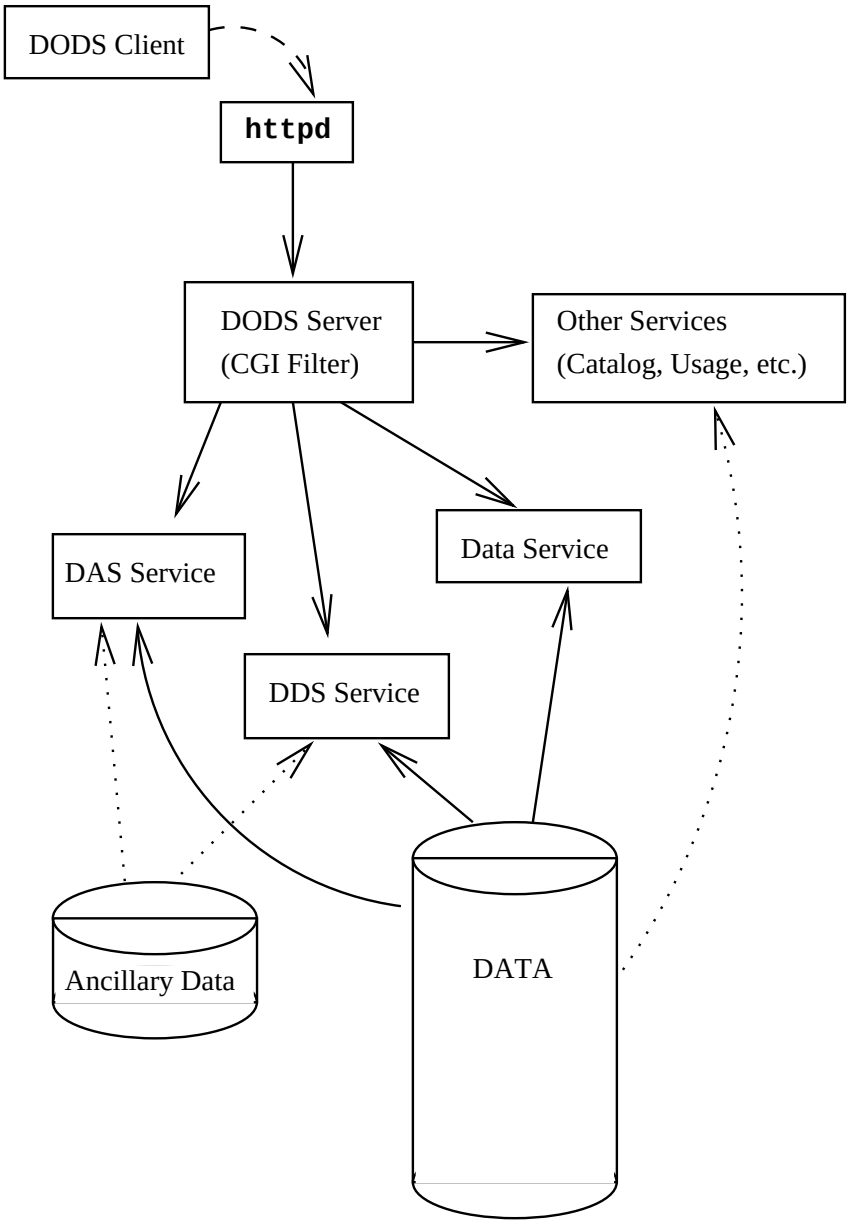

Figure 5.1: The Architecture of a DODS Data Server.

NOTE: There are other important DODS services. For a description of all the DODS services, see Section 2.2 on page 20.

The DODS server is structured as a dispatch function, invoking ancillary helper programs to provide its services. Installing a DODS server involves making sure that each of the required helper programs is available to the server software. Here is a table of the helper programs required for each of the DODS services for the netCDF server. For another DODS server, the names of some of the helper programs would have a different root (e.g. `ff_` for the FreeForm server, `jg_` for JGOFS, etc.).

Table 5.1: DODS Services, with their suffixes and helper programs.

Service	Suffix	Helper Program
Data Attribute	`.das`	`nc_das`
Data Descriptor	`.dds`	`nc_dds`
DODS Data	`.dods`	`nc_dods`
ASCII Data	`.asc` or `.ascii`	`asciival`
Information	`.info`	`usage`, see Section 5.2.3 on page 56 for configuration information.
WWW Interface	`.html`	None
Version	`.ver`	None
Help	Anything else	None

The service programs are started by the CGI depending on the extension given with the URL. If the URL ends with '.das' then the DAS service program is started. Similarly, the extension '.dds' will cause the DDS service to run and so on. The CGI program (the "dispatch" script), which serves to dispatch the request to one of the three service programs, can be very simple. In the servers distributed with DODS, the CGI is simply a shell script that takes its own name and catenates the enclosed URL suffix. The services, being more complex programs, will generally be written in C or C++.

On the client side, the user may never see the '.das,' '.dds,' or '.dds' URL extensions. Nor will the user necessarily be aware that each URL given to the DODS client produces three different requests for information. These manipulations happen within the client library, and the user need never be aware of them.

There may be more than five service programs for a given server implementation.[2] A server may provide other "services," such as the catalog service, or a service specific to a particular data implementation. The three data services, however, constitute the minimum configuration for a functional server. All three services are involved in data requests, as the client program will use the output from the `_dds` and `_dds` services to allocate memory and define parameters for the output of the `_dods` service, which is the actual data requested. The remaining two services, the ASCII and information services, are primarily intended for interactive use, as they make dataset and service information directly available to a browser client, such as Netscape.

[2] A couple of services, such as the version and help services, are built into the server software, and need no configuration.

5.2 Installing a DODS Server

Most of the task of installing a DODS server consists of getting the required Web server installed and running. The intricacies of this task, and the variety of available Web servers make this task beyond the scope of this guide. Proceed with the following steps only after the Web server itself is operational.

Installing the DODS CGI programs and the data to be served is a relatively simple operation. After installing the DODS source tree and building the software, (See Appendix A on page 91), the user need only copy the CGI program from the `etc` directory in the DODS source tree (`$(DODS_ROOT)/etc`) to one of the directories where the Web server expects to find its CGI programs. The exact name of this directory is an implementation detail of the Web server itself.

The service programs used by the CGI are generally kept in the same directory as the CGI itself, although this can be changed by modifying the DODS CGI dispatch script.

NOTE: The server programs come with release notes and installation notes, in files `README` and `INSTALL`, among others. These will be found in the distribution directories for the particular server. For example, the documentation for the JGOFS server will be found in `$DODS_ROOT/src/http/jg-dods`. See *The DODS Toolkit Programmer's Guide* for additional information about server documentation.

After installing the CGI program and the services, the data to be provided must be put in some location where it may be served to clients. Again, the location of the data depends on the configuration of the Web server and the API used by the CGI services. Most often, data that is served by a Web server is kept in the `htdocs` directory, the exact pathname of which is specified in the `httpd`.configuration file. A server may also be enabled to search a user's home directory tree or may follow links from the `htdocs` directory (if the server is enabled to follow symbolic links). There may be yet other options provided by the specific server used in a particular installation, so there is really no way to avoid consulting the configuration instructions of the Web server.

As noted, the location of the data depends not only on the configuration of the Web server, but also on the API used to access the data requested. For example, the netCDF server simply stores data in a path relative to the working directory of the CGI program, `htdocs`, while the JGOFS server uses its data dictionary to specify the location of its data. Refer to the specific installation notes for each API for more information about the location of the data.

5.2.1 Configuring the Server

The issues of server configuration depend to a large extent on the particular server in question. The DODS server for JGOFS data is configured differently than the DODS server for netCDF data. Each server comes with its own installation and configuration instructions. These can be found in a file called `INSTALL` in the distribution directory for the server. The server distribution directories are in `$DODS_ROOT/src`. Here is a checklist of items that need to be attended in order to install any DODS server:

- Is the `httpd` server configured to execute CGI programs?
- Are the main CGI and subsidiary CGI programs installed in the server's CGI directory? For the netCDF API, these will be called `nph-nc`, and `nc_das`, `nc_dds`, and so on. The server CGI's for other API's will have comparable names.
- Is the `gzip` program installed in the `PATH` of the `httpd` server? This is used to compress data messages returned to the client.

5.2.2 Constructing the URL

After a dataset has been installed, and the server programs installed, you need to know what its address is. Section 2.1 on page 18 contains an explanation of the various parts of the DODS URL, including a diagram in figure 2.1. Refer to this section, with a copy of the Web server configuration data readily available. Using the configuration data, you should be able to determine the appropriate URL for the data you are serving.

Remember that the web server will have its own definition of the root directory for data, and another definition for CGI programs, depending on the configuration.

5.2.3 Documenting Your Data

DODS contains provisions for supplying documentation to users about a server, and also about the data that server provides. When a server receives an information request (through the `info` service that invokes the `usage` program), it returns to the client an HTML document created from the DAS and DDS of the referenced data. It may also return information about the server, and more detail about the dataset.

If you would like to provide more information about a dataset than is contained in the DAS and DDS, simply create an HTML document (without the `<html>` and `<body>` tags, which are supplied by the `info` service), and store it in the same directory as the dataset, with a name corresponding to the dataset filename. For

example, the datasets `fnoc1.nc`, `fnoc2.nc`, and `fnoc3.nc` might be documented with a file called `fnoc.html`.

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.

More information about providing user information, including sample HTML files, and a complete description of the search procedure for finding the dataset documentation, is to be found in *The DODS Toolkit Programmer's Guide*.

5.2.4 Testing the Installation

It is possible to test the DODS server to see whether an installation has been properly done. The easiest way to test the installation is with a simple Web client like Netscape or Mosaic. (A simple Web client called `geturl` is provided in the DODS core software which can retrieve text from Web servers. Look for it in the `$(DODS_ROOT)/etc` directory.)

The simplest test is simply to ask for the version of the server, or the help message. The DODS server uses helper programs to return the DAS, DDS, and data. If you want to test the server itself, and not the configuration of the helper programs, the version, help, or info services will suffice. Issuing a URL with `.ver` on the end will return the version information for this server, appending `.info` will return the info message, and issuing a URL with a nonsense suffix or `.help` will return a help message:

```
> geturl http://dods.gso.uri.edu/cgi-bin/nph-nc/data/test.nc.ver
> geturl http://dods.gso.uri.edu/cgi-bin/nph-nc/data/test.nc.info
> geturl http://dods.gso.uri.edu/cgi-bin/nph-nc/data/test.nc.help
```

To return the data attribute structure of a dataset, use a URL such as the following:[3]

```
> geturl http://dods.gso.uri.edu/cgi-bin/nph-nc/data/test.nc.das
```

Refer to Section 6.4.2 on page 82 for a description of a data attribute structure. You can compare the description against what is returned by the above URL to test the operation of the DODS server.

You can use your web client to test the DODS server by using it to submit URLs that address specific services of the client. See Section 2.2 on page 20 for information about how to request individual services. If any of the services fail, you can check the list of helper programs in Section 5.1.1 on page 51 to find out

[3]The **`geturl`** program knows about the DODS protocols, so you can also omit the **`.das`** suffix, and use the **`-a`** option to the **`geturl`** command. This tells **`geturl`** to append **`.das`** for you.

which is missing. From the web browser, you can access all the DODS services, except the (binary) data service. However, if all the others work, you can be relatively assured that one will, too.

Using the `.html` suffix produces the WWW Interface , providing a forms-based interface with which a user can query the dataset using a simple web browser. There's more about the WWW Interface in Chapter 3.

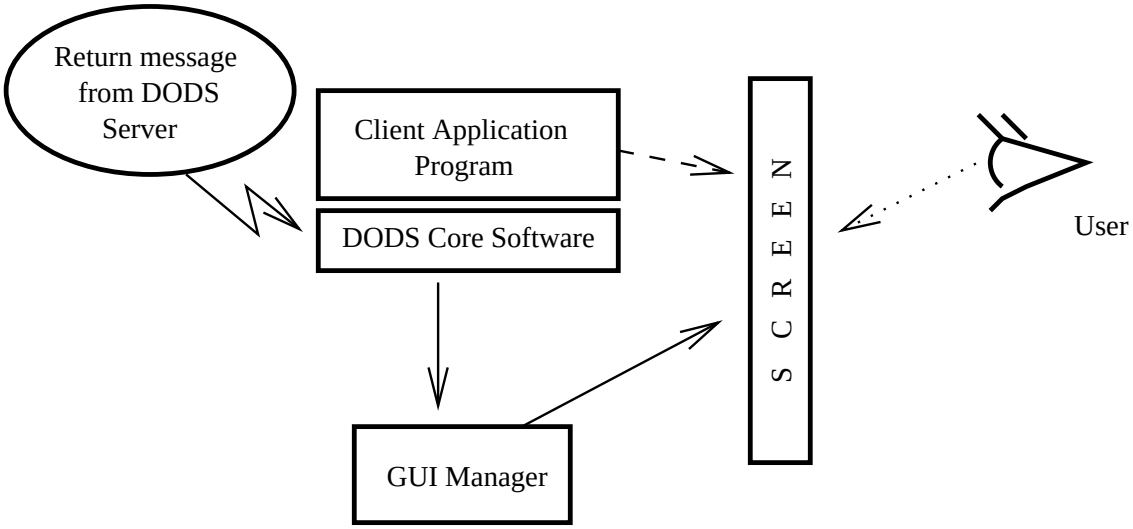

Figure 5.2: The Architecture of a DODS Client GUI.

5.3 Displaying Information to the DODS User

DODS contains a system that allows a DODS server a degree of control over the user's graphic user interface (GUI). This system runs the system progress indicator, that displays to the user the status of a pending data request. However, a server may also use the GUI interface to display messages to the user, such as error messages, and even to query the user for information.

5.3.1 GUI Architecture

Since DODS is built inside a data access API, and since the application program that has become the DODS client was presumably not built with network I/O in mind, a DODS client will typically not do any processing at all while it awaits a return message from a data request. Any communication that must happen between the DODS software and the user must occur without the involvement of the application program that has invoked the DODS software. To avoid this limitation, DODS starts up a *GUI manager* sub-process. This sub-process can receive data from the DODS core software, and can operate the user's graphical user interface.

The operation of the GUI manager is illustrated in figure 5.2. As seen in the figure, the client application can usually control the user's screen, but during a data request, this communication is suspended. Until the request returns control to the client application, messages returned from the DODS server can be displayed

to the user by passing them to the GUI manager sub-process, who can display them in a window to the user.

The GUI manager in 3.2uses a Tcl/Tk interpreter (the `wish` program is the default) to interpret messages from the server. These messages usually contain Tcl programs to display information to the user. However, the `wish` interpreter can also be sent programs to query the user for more information, or draw little rabbits on the screen or any other graphic function the server needs to have displayed to the user. See Tcl and the Tk Toolkit [3] for more information about Tcl.

By default, the GUI manager initializes by running the Tcl programs in the files `dods_gui.tcl`, `error.tcl` and `progress.tcl`. (These are stored in `$DODS_ROOT/etc`.) Server commands to the GUI manager can use the functions defined in these files. Note also that the user may be using a "safe" Tcl interpreter, with a restricted subset of the usual array of Tcl commands available to it. The user can control these features of the operation of the GUI by changing several environment variables. These are described in Section 2.3.2 on page 24.

A server will use the features of the GUI manager to display error messages to the user. A server may also use the GUI to query a user to correct whatever condition caused the error. For example, if a user has misspelled some part of a constraint expression in a URL submitted to a server, the server can send the constraint expression back to the user in an edit window, with instructions to fix it. The user can edit the expression, and send it back, allowing the server to proceed without submitting a new request. Consult the client and server toolkit manual for more information about the *Error* object on this subject.

5.4 Building DODS Data Servers

Though servers are included in the DODS core software, some users may wish to write their own DODS data servers. The architecture of the `httpd` server and the DODS core software make this a relatively simple task.

A user may wish to write his or her own DODS server for any or all of the following reasons:

- The data to be served may be stored in a format not compatible with one of the existing DODS servers.
- The data may be arranged in a fashion that allows a user to optimize the access of those data by rewriting the service programs.
- The user may wish to provide ancillary data to DODS clients not anticipated by the writers of the servers available.

The design of the DODS library make the task a relatively simple one for a programmer already familiar with the data access API to be used. Also, though the servers provided with the DODS core software are written in C++, they may be written in any language from which the DODS libraries may be called.

Once it is invoked, a CGI program scoops up whatever input is going to the standard input stream of the Web server (`httpd`) that invoked it. Further, the standard output of the CGI is piped directly to the WWW library, which sends it directly back to the requesting client. This means that the CGI program itself need only read its input from standard input and write its output to standard output.

Most of the task of writing a server, then, consists of reading the data with the data access API and loading it into the DODS classes. Method functions defined for each class make it simple to output the data so that it may be sent back to the requesting client.

Refer to *The DODS Toolkit Programmer's Guide* for specific information about the classes and the facilities of the DODS core software, and instructions about how to write a new server.

Part IV

Technical Documentation

6
Data and Data Models

Basic to the operation of DODS is the translation of data from one format to another. A DODS server must read data on some disk and translate it into an intermediate format for transmission to the client. It is to the question of these formats to which we shall turn first.

6.1 Data models

Any data set is made up of data and a *data model* . The data model defi nes the size and arrangement of data values, and may be thought of as an abstract representation of the relationship between one data value and another. Though it may seem paradoxical, it is precisely this relationship that defi nes the meaning of some number. Without the context provided by a data model, a number does not represent anything. For example, within some data set, it may be apparent that a number represents the value of temperature at some point in space and time. Without its neighboring temperature measurements, and without the latitude, longitude, depth, and time, the same number means nothing.

As the model only defi nes an abstract set of relationships, two data sets containing different data may share the same data model. For example, the data produced by two different measurements with the same instrument will use the same data model, though the values of the data are different. Sometimes two models may be equivalent. For example, an XBT measures a time series of temperature, but is usually stored as a series of temperature and depth measurements. The temperature vs. time model of the original data is equivalent to the temperature vs. depth model of the stored data.

In a computational sense, a data model may be considered to be the data type or collection of data types used to represent that data. A temperature measurement might occur as half an entry in a sequence of temperature and depth pairs. However the data model also includes the scalar latitude, longitude and date that identify the time and place where the temperature measurements were taken. Thus the data set might be represented in a C-like syntax like this (fi gure 6.1):

```
Dataset {
   Float64 lat;
   Float64 lon;
   Int32 minutes;
   Int32 day;
   Int32 year;
   Sequence {
      Float64 depth;
      Float64 temperature;
   } cast;
} xbt-station;
```

Figure 6.1: Example Data Description of XBT Station

In the above example, a data set is described that contains all the data from a single XBT. The data set is called `xbt-station`, and contains floating-point representations of the latitude and longitude of the station, and three integers that specify when the XBT was released. The `xbt-station` contains a single

sequence (called `cast`) of measurements, which are here represented as values for depth and temperature[1].

A different data model representing the same data might look like this (figure 6.2):

```
Dataset {
   Structure {
      Float64 lat;
      Float64 lon;
   } location;
   Structure {
      Int32 minutes;
      Int32 day;
      Int32 year;
   } time;
   Sequence {
      Float64 depth;
      Float64 temperature;
   } cast;
} xbt-station;
```

Figure 6.2: Example Data Description of XBT Station Using Structures

In this example, several of the data have been grouped, implying a relation between them. The nature of the relationship is not defined, but it is clear that `lat` and `lon` are both components of `location`, and that each measurement in the `cast` sequence is made up of depth and temperature values.

In these two examples, meaning was added to the data set only by providing a more refined context for the data values. No other data was added, but still the second example can be said to contain more information than the first one.

These two examples are refinements of the same basic arrangement of data. However, there is nothing that says that a completely different data model can't be just as useful or just as accurate. For example, the depth and temperature data, instead of being represented by a sequence of pairs, as in figure 6.1 and figure 6.2, could be represented by a pair of sequences or arrays, as in figure 6.3

The relationship between the depth and temperature variables is no longer clear, but, depending on what sort of processing is intended, this may not be that important a loss.

The choice of a computational data model to contain some data set depends in many cases on the whims and preferences of the user, as well as on the data analysis software to be used. Several different data models may be equally useful

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

```
Dataset {
   Structure {
      Float64 lat;
      Float64 lon;
   } location;
   Structure {
      Int32 minutes;
      Int32 day;
      Int32 year;
   } time;
   Float64 depth[500];
   Float64 temperature[500];
} xbt-station;
```

Figure 6.3: Example Data Description of XBT Station Using Arrays

for a given task. Of course, some data models will contain more information about the data than others, but this information can also be carried in a scientist's head.

Note that with a carefully chosen set of data type constructors , such as those we've used in the preceding examples, a user can implement an infi nite number of data models. The examples above use the DODS Dataset Descriptor Structure (DDS) format, which will become important in later discussions of the details of the DODS Data Access Protocol. The precise details of the DDS syntax are described in Section 6.4.1 on page 81.

6.1.1 Data Models and APIs

A data access Application Program Interface (API) is a library of functions designed to be used by a computer program to read, write, and sample data. Any given data access API can be said to defi ne implicitly some data model. That is, the functions that make up the API accept and return data using a certain collection of computational data types: multi-dimensional arrays might be required for some data, scalars for others, lists for others. This collection of data types, and their use constitute the data model represented by that API. (Or data models—there is no reason an API cannot accommodate several different models.)

Among others, DODS currently supports two very different data access APIs: netCDF and JGOFS. The netCDF API is designed for access to gridded data, but has some limited capacity to access sequence data. The JGOFS API provides access to relational or sequence data. Both APIs support access in several programming languages (at least C and Fortran) and both provide extensive support for limiting the amount of data retrieved. For example a program accessing a gridded dataset using netCDF can extract a subsampled portion or *hyperslab* of that data. Likewise, the JGOFS API provides a powerful set of

operators which can be used to specify which sequence elements to extract (for example, a user could request only those values corresponding to data captured between 12:01am and 11:59am) as well as masking certain parameters from the returned elements so that only those parameters needed by the program are returned.

6.1.2 Translating Data Models

The problem of data model translation is central to the implementation of DODS. With an effective data translator, a DODS program originally designed to read netCDF data can have some access to data sets that use an incompatible data model, such as JGOFS.

In general, it is not possible to define an algorithm that will translate data from any model to any other, without losing information defined by the position of data values or the relations between them. Some of these incompatibilities are obvious; a data model designed for time series data may not be able to accommodate multi-dimensional arrays. Others are more subtle. For example, a sequence looks very similar to a collection of lists in many respects. However, a sequence is an *ordered* collection of data types, whereas a list implies no order. However, there are many useful translations that can be done, and there are many others that are still useful despite their inherent information loss.

For example, consider a relational structure like the one in figure 6.4. This is similar to the examples in Section 6.1 on page 66, rearranged to accommodate an entire cruise worth of temperature-depth measurements. This is the sort of data type that the JGOFS API is designed to use.

```
Dataset {
   Sequence {
      Int32 id;
      Float64 latitude;
      Float64 longitude;
      Sequence {
         Float64 depth;
         Float64 temperature;
      } xbt_drop;
   } station;
} cruise;
```

Figure 6.4: Example Data Description of XBT Cruise

Note that each entry in the `cruise` sequence is composed of a tuple of data values (one of which is itself a sequence). Were we to arrange these data values as a table, they might look like this:

id	lat	lon	depth	temp
1	10.8	60.8	0	70
			10	46
			20	34
2	11.2	61.0	0	71
			10	45
			20	34
3	11.6	61.2	0	69
			10	47
			20	34

This can be made into an array, although that introduces redundancy.

id	lat	lon	depth	temp
1	10.8	60.8	0	70
1	10.8	60.8	10	46
1	10.8	60.8	20	34
2	11.2	61.0	0	71
2	11.2	61.0	10	45
2	11.2	61.0	20	34
3	11.6	61.2	0	69
3	11.6	61.2	10	47
3	11.6	61.2	20	34

The data is now in a form that may be read by an API such as netCDF. But consider the analysis stage. Suppose a user wants to see graphs of station data. It is not obvious simply from the arrangement of the array where a station stops and the next one begins. Analyzing data in this format is not a function likely to be accommodated by a program that uses the netCDF API.

6.2 Data Access Protocol

The DODS Data Access Protocol (DAP) defines how a DODS client and a DODS server communicate with one another to pass data from the server to the client. The job of the functions in the DODS client library is to translate data from the DAP into the form expected by the data access API for which the DODS library is substituting. The job of a DODS server is to translate data stored on a disk in whatever format they happen to be stored in to the DAP for transmission to the client.

The DAP consists of four components:

❶ An *intermediate data representation* for data sets. This is used to transport data from the remote source to the client. The data types that make up this representation may be thought of as the DODS data model.

❷ A format for the *ancillary data* needed to translate a data set into the intermediate representation, and to translate the intermediate representation into the target data model. The ancillary data in turn consists of two pieces:

- ➨ A description of the shape and size of the various data types stored in some given data set. This is called the *Data Description Structure* (DDS).
- ➨ Capsule descriptions of some of the properties of the data stored in some given data set. This is the *Data Attribute Structure* (DAS).

❸ A *procedure* for retrieving data and ancillary data from remote platforms.

❹ An *API* consisting of DODS classes and data access calls designed to implement the protocol,

The intermediate data representation and the ancillary data formats are introduced in Section 6.3 on page 72 and Section 6.4 on page 81, below. The steps of the procedure are outlined in Section 5.1 on page 50, and the DODS core software is described in the *The DODS Toolkit Programmer's Guide*.

6.3 Data representation

There are many popular data storage formats, and many more than that in use. These formats are optimized (it they are optimized at all) for data storage, and are not generally suitable for data transmission. In order to transmit data over the Internet, DODS must translate the data model used by a particular storage format into the data model used for transmission.

If the data model for transmission is defined to be general enough to encompass the abstractions of several data models for storage, than this intermediate representation–the transmission format –can be used to translate between one data model and another.

The DODS data model consists of a fairly elementary set of base types, combined with an advanced set of constructs and operators that allows it to define data types of arbitrary complexity. This way, the DODS data access protocol can be used to transmit data from virtually any data storage format.

The elements of the DODS data access protocol are:

- **Base Types** These are the simple data types, like integers, floating point numbers, strings, and character data.
- **Constructor Types** These are the more complex data types that can be constructed from the simple base types. Examples are structures, sequences, arrays, and grids.
- **Operators** Access to data can be operationally defined with operators defined on the various data types.
- **External Data Representation** In order to transmit the data across the Internet, there needs to be a machine-independent definition of what the various data types look like. For example, the client and server need to agree on the most significant digit of a particular byte in the message

These elements are defined in greater detail in the sections that follow.

6.3.1 Base Types

The DODS data model uses the concepts of variables and operators. Each data set is defined by a set of one or more variables, and each variable is defined by a set of attributes. A variable's *attributes*—such as units, name and type—must not be confused with the data *value* (or values) that may be represented by that variable. A variable called `time` may contain an integer number of minutes, but it does not contain a particular number of minutes until a context, such as a specific event

recorded in a data set, is provided. Each variable may further be the object of an operator that defines a subset of the available data set. This is detailed in Section 6.3.3 on page 77.

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

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

Type The data type contained by the variable. This can be one of `Byte`, `Int32`, `UInt32`, `Float64`, `String`, and `URL`. Where:

- `Byte` is a single byte of data. This is the same as `unsigned char` in ANSI C.
- `Int32` is a 32 bit two's complement integer—it is synonymous with long in ANSI C when that type is implemented as 32 bits.
- `UInt32` is a 32 bit unsigned integer.
- `Float64` is the IEEE 64 bit floating point data type.
- `String` is a sequence of bytes terminated by a null character.
- `Url` is a string containing a DODS URL. Please refer to Section 2.1 on page 18 for more information about these strings. A special `*` operator is defined for a URL. If the variable `my-url` is defined as a URL data type, then `my-url` indicates the string spelling out the URL, and `*my-url` indicates the data specified by the URL.

The declaration in a DDS of a variable of any of the base types is simply the type of the variable, followed by its name, and a semicolon. For example, to declare a `month` variable to be a 32-bit integer, one would type:

```
Int32 month;
```

6.3.2 Constructor Types

Constructor types, such as arrays, structures, and lists, describe the grouping of one or more variables within a dataset. These classes are used to describe different types of relations between the variables that comprise the dataset. For example, an array might indicate that the variables grouped are all measurements of the same quantity with some spatial relation to one another, whereas a structure might indicate a grouping of measurements of disparate quantities that happened at the same place and time.

There are six classes of type constructor variables defined by the DODS DAP: lists, arrays, structures, sequences, functions, and grids. The types are defined as:

List The list type constructor is used to hold lists of 0 or more items of one type. Lists are specified using the keyword `list` before the variable's class, for example, `list int32` or `list grid`. Access to the elements of a list is possible using one of the three operators shown in table 6.1:

list.`length` Returns the integer length of the *list*.

list.`nth(`*n*`)` Returns the *n*th member of the *list*.

list.`member(`*value*`)` Returns `true` if the *value* is a member of the *list*.

> **NOTE:** The syntax of these operators differs between their use in a C++ program and a constraint expression. The length of some list, given by `list.length()` in a program, would be `length(list)` in a constraint expression. Similarly, in a constraint expression, the position of a value in a list is given by `nth(list, value)`, and the presence of a value is indicated by `member(list, value)`. See Section 4.1 on page 36 for more information about constraint expressions.

A list declaration to create a list of integers would look like the following:

```
List Int32 months;
```

Array An array is a one dimensional indexed data structure as defined by ANSI C. Multidimensional arrays are defined as arrays of arrays. An array may be subsampled using subscripts or ranges of subscripts enclosed in brackets (`[]`). For example, `temp[3][4]` would indicate the value in the fourth row and fifth column of the `temp` array.[2] A chunk of an array may be specified with subscript ranges; the array `temp[2:10][3:4]` indicates an array of nine rows and two columns whose values have been lifted intact from the larger `temp` array.

A *hyperslab* may be selected from an array with a *stride* value. The array represented by `temp[2:2:10][3:4]` would have only five rows; the middle value in the first subscript range indicates that the output array values are to be selected from alternate input array rows. The array `temp[2:3:10][3:4]` would select from every third row, and so on. table 6.1 shows the syntax for array accesses including hyperslabs.

To declare a $5x6$ array of floating point numbers, the declaration would look like the following:

```
Float64 data[5][6];
```

[2] As in C, DODS array indices start at zero.

In addition to its magnitude, every dimension of an array may also have a name. The previous declaration could be written:

```
Float64 data[height = 5][width = 6];
```

Structure A Structure is a class that may contain several variables of different classes. However, though it implies that its member variables are related somehow, it conveys no relational information about them. The structure type can also be used to group a set of unrelated variables together into a single dataset. The `dataset` class name is a synonym for `structure`.

A Structure declaration containing some data and the month in which the data was taken might look like this:

```
Structure {
   Int32 month;
   Float64 data[5][6];
} measurement;
```

Use the . operator to refer to members of a *Structure*. For example, `measurement.month` would identify the integer member of the *Structure* defined in the above declaration.

Sequence A *Sequence* is an ordered set of variables each of which may have several values. The variables may be of different classes. Each element of a *Sequence* consists of a value for each member variable. Thus a *Sequence* can be represented as:

$$\begin{matrix} s\_00 & \cdots & s\_0n \\ \vdots & \ddots & \vdots \\ s\_i0 & \cdots & s\_in \end{matrix}$$

Every instance of sequence S has the same number, order, and class of member variables. A *Sequence* implies that each of the variables is related to each other in some logical way. For example, a sequence containing position and temperature measurements might imply that the temperature measurements were taken at the corresponding position. A sequence is different from a structure because its constituent variables have several instances while a structure's variables have only one instance (or value). Because a sequence has several values for each of its variables it has an implied *state* , in addition to those values. The state corresponds to a single element in the sequence.

A *Sequence* declaration is similar to a *Structure*'s. For example, the following would define a *Sequence* that would contain many members like the *Structure* defined above:

```
Sequence {
   Int32 month;
   Float64 data[5][6];
} measurement;
```

Note that, unlike an *Array*, a *Sequence* has no index. This means that a *Sequence*'s values are not simultaneously accessible. Like a *Structure*, the variable `measurement.month` has a single value. The distinction is that this variable's value changes depending on the state of the *Sequence*.

Grid A *Grid* is an association of an *N* dimensional array with *N* named vectors (one-dimensional arrays), each of which has the same number of elements as the corresponding dimension of the array. Each data value in the grid is associated with the data values in the vectors associated with its dimensions.

As an example, consider an array of temperature values that is six columns wide by five rows long. Suppose that this array represents measurements of temperature at five different depths in six different locations. The problem is the indication of the precise location of each temperature measurement, relative to one another.[3]

If the six locations are evenly spaced, and the five depths are also evenly spaced, then the data set can be completely described using the array and two scalar values indicating the distance between adjacent vertices of the array. However, if the spacing of the measurements is *not* regular, as in figure 6.5 then an array will be inadequate. To adequately describe the positions of each of the points in the grid, the precise location of each volume and row must be described.

The secondary vectors in the *Grid* data type provide a solution to this problem. Each member of these vectors defines a value for all the data values in the corresponding rank of the array. The value can represent location or time or some other quantity, and can even be a constructor data type. The following declaration would define a data type that could accommodate a structure like this:

```
Grid {
   Float64 data[distance = 6][depth = 5];
   Float64 distance[6];
   Float64 depth[5];
} measurement;
```

In the above example, an vector called `depth` would contain five values corresponding to the depths of each row of the array, while another vector called `distance` might contain the scalar distance between the location of the corresponding column, and some reference point. The `distance` array could also contain six (latitude, longitude) pairs indicating the absolute location of each column of the grid.

[3]The absolute location and orientation of the entire array is specified by another set of scalar values; we are here considering the relationship between data type members.

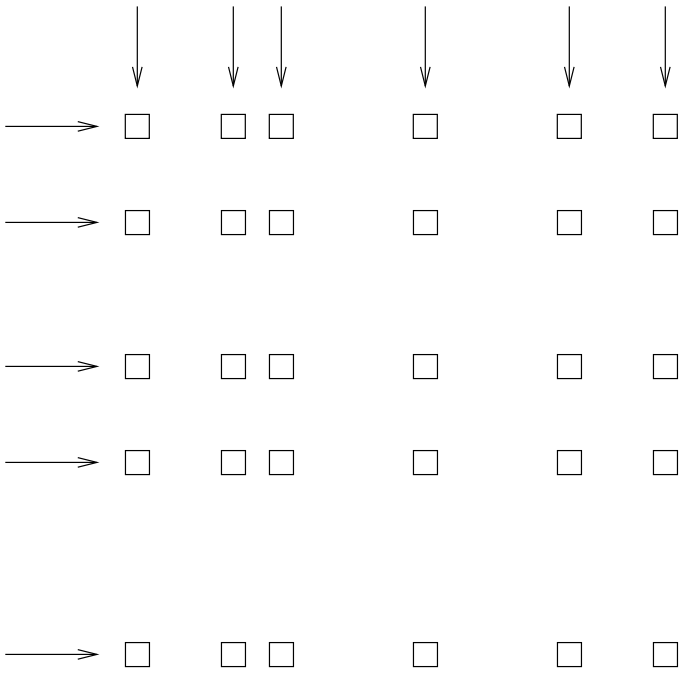

Figure 6.5: An Irregular Grid of Data.

```
Grid {
   Float64 data[distance = 6][depth = 5];
   Float64 depth[5];
   Array Structure {
      Float64 latitude;
      Float64 longitude;
   } distance[6];
} measurement;
```

6.3.3 Operators

Access to variables can be modified using operators. Each type of variable has its own set of selection and projection operators which can be used to modify the result of accessing a variable of that type. table 6.1 lists the types and the operators applicable to them. In the table, operators have the meaning defined by ANSI C except as follows: the array hyperslab operators are as defined by netCDF[4] , the string operators are as defined by AWK[2] , and the list operators are as defined by Common Lisp[5] .

Two of the operators deserve special note. Individual fields of type constructors may be accessed using the dot (`.`) operator or the virtual file system syntax. If a structure `s` has two fields `time` and `temperature`, then those fields may be accessed using `s.time` and `s.temperature` or as `s/time` and `s/temperature`. Also, a special dereferencing `*` operator is defined for a URL. This is roughly

Table 6.1: Classes and operators in the DAP.

Class	Operators
Simple Types	
Byte, Int32, UInt32, Float64	`< > = != <= >=`
String	`= != ~=`
URL	`*`
Compound Types	
Array	`[start:stop]` `[start:stride:stop]`
List	`length(`*list*`),` `nth(`*list*`,`*n*`),` `member(`*list*`,`*elem*`)`
Structure	`.`
Sequence	`.`
Grid	`[start:stop]` `[start:stride:stop] .`

analogous to the pointer-dereference operator of ANSI C. That is, if the variable `my-url` is defined as a URL data type, then `my-url` indicates the string spelling out the URL, and `*my-url` indicates the actual data indicated by the URL.

More information about variables and operators can be found in the discussion of constraint expressions in Section 4.1 on page 36.

6.3.4 External Data Representation

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

Constraint expressions do not affect *how* a base-type variable is transmitted from a client to a server; they determine *if* a variable is to be transmitted. For constructor type variables, however, constraint expressions may be used to exclude portions of the variable. For example, if a constraint expression is used to select the first three of six fields in a structure, the last three fields of that structure are not transmitted by the server.

The data access protocol uses Sun Microsystems' XDR protocol[6] for the

Table 6.2: The XDR data types corresponding to DODS base-type variables

Base Type	XDR Type
Byte	`xdr byte`
Int32	`xdr long`
UInt32	`xdr unsigned long`
Float64	`xdr double`
String	`xdr string`
URL	`xdr string`

external representation of all of the base type variables. table 6.2 shows the XDR types used to represent the various base type variables.

In order to transmit constructor type variables, the data access protocol defines how the various base type variables, which comprise the constructor type variables, are transmitted. Any constructor type variable may be subject to a constraint expression which changes the amount of data transmitted for the variable (see Section 4.1 on page 36 for more information about constraint expressions.). For each of the six constructor types these definitions are:

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

List A *List* is sent as if it were an *Array*.

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

Sequence A *Sequence* is transmitted by encoding each item in the sequence as if it were a *Structure*, and sending each such structure after the other, in the order of their occurrence in the sequence. The entire sequence is sent, subject to the constraint expression. In other words, if no constraint expression is supplied then the entire sequence is sent. However, if a constraint expression is given all the records in the sequence that satisfy the expression are sent[4].

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

The external data representation used by a DODS server and client may be compressed, depending on the configuration of the respective machines. The

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

compression is done using the `gzip` program. Only the data transmission itself will be affected by this; the transmission of the ancillary data is not compressed.

6.4 Ancillary data

In order to use some data set, a user must have some information at his or her disposal that is not strictly included in the data set itself. This information, called *ancillary data* (and sometimes called *metadata* [5]), describes the shape and size of the data types that make up the data set, and provides information about many of the data set's attributes, as well. DODS uses two different structures, to supply this ancillary information about a DODS data set. The Dataset Descriptor Structure (DDS) describes the data set's structure and the relationships between its variables, and the Dataset Attribute Structure (DAS) provides information about the variables themselves. Both structures are described in the following sections.

6.4.1 Dataset Descriptor Structure

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

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

The DDS is a textual description of the variables and their classes that make up some data set. The DDS syntax is based on the variable declaration and definition syntax of C and C++. A variable that is a member of one of the base type classes is declared by writing the class name followed by the variable name. The type constructor classes are declared using C's brace notation. A grammar for the syntax is given in table 6.3. Each of the keywords for the type constructor and base type classes have already been described in Section 6.3 on page 72. The `Dataset` keyword has the same syntactic function as *Structure* but is used for the specific job of enclosing the entire data set even when it does not technically need an enclosing element.

Different data access APIs will store the information in the DDS in different

[5] We have learned to shy away from this term since we have found that 'metadata' to one person is 'data' to another; the categorization often limits the usefulness of the underlying information.

Table 6.3: Dataset Descriptor Structure Syntax

data set	`Dataset` { *declarations* } *name* ;
declaration	`List` *declaration* *base-type var* ; `Structure` { *declarations* } *var* ; `Sequence` { *declarations* } *var* ; `Grid` { `ARRAY` : *declaration* `MAPS` : *declarations* } *var* ;

base-type	`Byte`
	`Int32`
	`UInt32`
	`Float64`
	`String`
	`Url`

var	*name* *name array-decl*
array-decl	[integer] [*name* = integer]
name	User-chosen name of data set, variable, or array dimension.

places. Some APIs are self-documenting in the sense that the data files themselves will contain all the information about the structure of their data types. Other APIs need secondary files containing what is called ancillary data , describing the data structure. For some APIs, such as netCDF, gathering the ancillary information from the data archive may be a time-consuming process. The DODS server for these APIs may cache ancillary data files to save time. An example DDS entry is shown in figure 6.6. (See Section 6.1 on page 66 for an explanation of the information implied by the data model, and for several other DDS examples).

When creating a DDS to be kept in an ancillary file, you can use the **#** character as a comment indicator. All characters after the **#** on a line are ignored.

6.4.2 Dataset Attribute Structure

The *Dataset Attribute Structure* (DAS) is used to store attributes for variables in the dataset. An attribute is any piece of information about a variable that the creator wants to bind with that variable *excluding* the type and shape, which are part of the DDS. Attributes can be as simple as error measurements or as elaborate as text describing how the data was collected or processed[6]. In principle, attributes are not processed by software, other than to be displayed. However, many systems rely on attributes to store extra information that is necessary to perform certain manipulations of data. In effect, attributes are used to store information that is used 'by convention' rather than 'by design'. DODS can

[6]To define attributes for the entire dataset, create an entry for a variable with the same name as the dataset.

```
Dataset {
  Int32 catalog_number;
  Sequence {
    String experimenter;
    Int32 time;
    Structure {
      Float64 latitude;
      Float64 longitude;
    } location;
    Sequence {
      Float64 depth;
      Float64 salinity;
      Float64 oxygen;
      Float64 temperature;
    } cast;
  } station;
} data;
```

Figure 6.6: Example Dataset Descriptor Entry.

effectively support these conventions by passing the attributes from data set to user program via the DAS. Of course, DODS cannot enforce conventions in datasets where they were not followed in the first place.

Similarly to the DDS, the actual location of the DAS storage will vary from one API to another. Data files created with some APIs will contain within themselves attribute information that can be contained in the DAS. For these APIs, the DAS will be constructed dynamically by the DODS server from data within the files.

Other data access APIs must have attribute information specified in an ancillary data file. APIs that contain attribute information can have that information enriched by the addition of these ancillary attribute files. These files are typically stored in the same directory as the data files, and given the same name as the data files, appended with `.das`.

The syntax for attributes in a DAS is given in table 6.4. Every attribute of a variable is a triple: attribute name, type and value. Note that the attributes specified using the DAS are different from the information contained in the DDS. Each attribute is completely distinct from the name, type, and value of its associated variable. The name of an attribute is an identifier, following the normal rules for an identifier in a programming language with the addition that the '/' character may be used. The type of an attribute may be one of: *Byte*, *Int32*, *UInt32*, *Float64*, *String* or *Url*. An attribute may be scalar or vector. In the latter case the values of the vector are separated by commas (,) in the textual representation of the DAS.

When creating a DAS to be kept in an ancillary file, you can use the **#** character as a comment indicator. All characters after the **#** on a line are ignored.

Table 6.4: Dataset Attribute Structure Syntax

DAS	`Attributes` { *var-attr-list* }
var-attr-list	*var-attr* *var-attr-list var-attr* (empty list)
var-attr	*variable* { *attr-list* } *container* { *var-attr-list* } *global-attr* *alias*
global-attr	`Global` *variable* { *attr-list* }
attr-list	*attr-triple* ; *attr-list attr-triple* *(empty list)*
attr-triple	*attr-type attribute attr-val-vec* ;
attr-val-vec	*attr-val* *attr-val-vec* , *attr-val*
attr-val	numeric value *variable* “string”

attr-type	`Byte`
	`Int32`
	`UInt32`
	`Float64`
	`String`
	`Url`

alias	`Alias` *alias-name variable* ;
variable	user-chosen variable name
attribute	user-chosen attribute name
container	user-chosen container name
alias-name	user-chosen alias name

Containers

An attribute can contain another attribute, or set of attributes. This is roughly comparable to the way compound variables can contain other variables in the DDS. The container defines a new lexical scope for the attributes it contains[7].

Consider the following example:

```
Attributes {
   Bill {
      String LastName "Evans";
      Byte Age 53;
      String DaughterName "Matilda";
      Matilda {
         String LastName "Fink";
         Byte Age 26;
      }
   }
}
```

Figure 6.7: An Example of Attribute Containers

Here, the attribute `Bill.LastName` would be associated with the string "Evans", and `Bill.Age` with the number 53. However, the attribute `Bill.Matilda.LastName` would be associated with the string "Fink" and `Bill.Matilda.Age` with the number 26.

Using container attributes as above, you can construct a DAS that exactly mirrors the construction of a DDS that uses compound data types, like *Structure* and *Sequence*. Note that though the `Bill` attribute is a container, it has attributes of its own, as well. This exactly corresponds to the situation where, for example, a *Sequence* would have attributes belonging to it, as well as attributes for each of its member variables. Suppose the sequence represented a single time series of measurements, where several different data types are measured at each time. The sequence attributes might be the time and location of the measurements, and the individual variables might have attributes describing the method or accuracy of that measurement.

Aliases

Building on the previous example, it might be true that it would be convenient to refer to Matilda without prefixing every reference with `Bill`. In this case, we can define an *alias* attribute as follows:

[7] Containers, aliases, and global attributes were introduced into DODS at version 2.16. In early DODS releases, the DAS was *not* a hierarchical structure; it was similar to a flat-file database. Although using the new structure is strongly recommended for new code, old code will still work with the old DAS. See *The DODS Toolkit Reference* for a description of the changes made to the *AttrTable* class.

```
Attributes {
   Bill {
      String LastName "Evans";
      Byte Age 53;
      String DaughterName "Matilda";
      Matilda {
         String LastName "Fink";
         Byte Age 26;
      }
   }
   Alias Matilda Bill.Matilda;
}
```

Figure 6.8: An Example of Attribute Alias

By defining an equivalence between the alias `Matilda` and the original attribute `Bill.Matilda`, the string `Matilda.Age` can be used with or without the prefix `Bill`. In either case, the attribute value will be 26.

Global Attributes

A *global attribute* is not bound to a particular identifier in a dataset; these attributes are stored in one or more containers with the name `Global` or ending with `_Global`. Global attributes are used to describe attributes of an entire dataset. For example, a global attribute might contain the name of the satellite or ship from which the data was collected. Here's an example:

```
Attributes {
   Bill {
      String LastName "Evans";
      Byte Age 53;
      String DaughterName "Matilda";
      Matilda {
         String LastName "Fink";
         Byte Age 26;
      }
   }
   Alias Matilda Bill.Matilda;
   Global {
      String Name "FamilyData";
      String DateCompiled "11/17/98";
   }
}
```

Figure 6.9: An Example of Global Attributes

Global attributes can be used to define a certain view of a dataset. For example, consider the following DAS:

```
Attributes {
   CTD {
      String Ship "Oceanus";
      Temp {
         String Name "Temperature";
      }
      Salt {
         String Name "Salinity";
      }
   }
   Global {
      String Names "DODS";
   }
   FNO_Global {
      String Names "FNO";
      CTD {
         Temp {
            String FNOName "TEMPERATURE";
         }
         Salinity {
            String FNOName "SALINITY";
         }
      }
      Alias T CTD.Temp;
      Alias S CTD.Salt;
   }
}
```

Figure 6.10: An Example of Global Attributes In Use

Here, a dataset contains temperature and salinity measurements. To aid processing of this dataset by some DODS client, long names are supplied for the `Temp` and `Salt` variables. However, a different client (FNO) spells variable names differently. Since it is seldom practical to come up with general-purpose translation tables[8], the dataset administrator has chosen to include these synonyms under the `FNO_Global` attributes, as a convenience to those users.

Similar conveniences can be provided using the Alias feature. In the example in figure 6.10, the temperature variable can be referred to as `FNO_Global.T` if desired. That is, a global alias can provide a client with a known attribute name to query for some property, even if that attribute name is not an integral part of the dataset.

[8] "Temperature" can be spelled "T", "Temp", "TEMPERATURE", "TEMP", and so on. Worse, "T" is also commonly used for "Time."

Using global attributes, a dataset or catalog administrator can create a layer of aliases and attributes to make DODS datasets conform to several different dataset naming standards. This becomes signifi cant when trying to compile a DODS dataset database.

Part V

Appendices

A

Installing the DODS Software

The current version of the Distributed Oceanographic Data System core software is 3.2. Note that this number applies only to the core software. The individual servers, clients, and client libraries have their own version numbers.

Full information about the latest versions of the DODS software is available from the DODS web site: DODS Home page.

This version of DODS uses CGI standard 1.1.1.

A.1 Acquiring the DODS Software

The DODS Web Site

We recommend that you use the DODS Home page to obtain the most current version of the software. The "Software" page includes a form for selecting DODS components. Completing the form automatically creates a custom compressed archive of all the software components you selected, which you can then download to your own machine.

Whenever possible, you should use the provided binary software rather than trying to compile and link DODS yourself. Compiling will work, but the DODS software is large, and it takes a long time to compile.

Don't forget to select the core software as well as the specific data access API or server you need. For a DODS client, you will also need `Tcl/Tk` and `gzip`.

By Anonymous FTP

The DODS software may be down-loaded by anonymous `ftp` from the DODS ftp site.

Don't forget to down-load the core software `tar` file as well as the `tar` file corresponding to whatever data access API you need.

The DODS project provides a small number of archive files containing linked libraries and executables for some computing platforms. You should use these files whenever possible, to avoid the hassle of compiling the software yourself. The DODS software is a substantial chunk of code, and it takes a long time to compile and link.

A.2 Installing the Software

To install the DODS software, choose a directory to be the DODS root directory. This directory must be identified with the `DODS_ROOT` environment variable for the DODS software to run.

First, set the working directory to the `DODS_ROOT` directory.For example,

```
export DODS_ROOT=/usr/local/DODS
cd $DODS_ROOT
```

Next, expand the archives with `gzip` and unpack the expanded files with `tar`:

```
gzip -d DODS-88772.tar.gz
tar -xvf DODS-88772.tar
```

If you got the files via anonymous ftp, you would have to repeat the process for each down-loaded archive file, specifying the name of each component file. For example:

```
gzip -d DODS-core-2.19.tar.gz
tar -xvf DODS-core-2.19.tar
```

Building the DODS Core

Unpacking the core software archive will create a configure script in the `DODS_ROOT` directory. The core software may then be built with:

```
cd $DODS_ROOT
./configure
make
```

If you downloaded binary files, you can skip the "building" steps.

Building a DODS Client or Server

Unpacking the client library archives in the `DODS_ROOT` directory will produce directories such as `jg-dods` for the JGOFS software and `nc-dods` for the netCDF software. These will appear in the `src` subdirectory under `DODS_ROOT`. Each of these directories will also be equipped with a configure script to create a makefile, so the build procedure for them is the same as for the core software, for example:

```
cd $DODS_ROOT/src/nc-dods
./configure
make
```

Getting the Software Used by DODS

Several pieces of software are required to run DODS. These are all free software, and are descibed in Appendix B on page 99.

A.2.1 Installing the DODS Libraries

In order to link and run client programs, the only DODS software that must be installed are the DODS libraries. The following libraries, described in Section 3.1 on page 31 must be installed in the `/usr/lib` directory, or somewhere else where the linker will find them. Simply copy them into that directory from the DODS distribution.

- `libdap++.a`
- The DODS version of the API your software uses.

In addition to the DODS libraries, the following software is also required to link a DODS client. You can obtain them from the DODS Home page or DODS ftp site. (Look under your machine type, then under `DODS/packages/lib`.) The first two libraries should be part of the Tcl/Tk distribution. The other three libraries are GNU software.

- `libtcl.a`
- `libexpect.a`
- `libstdc++.a`
- `librx.a` (Regular expression software. Part of the `regex` package.)
- `libz.a` (Compression software. Part of the `gzip` distribution.)

To run a DODS client, you should have the `wish` and `gzip` programs in the `PATH`, and the following Tcl programs must be either in the `DODS_ROOT/etc` directory:

- `dod_gui.tcl`
- `progress.tcl`
- `error.tcl`

A DODS client will still execute without these programs, but important functionality, including the GUI manager and the data compression will be absent. Refer to the `INSTALL` file included in each distribution for specific information about setting up the client.

A.3 The DODS Client Initialization File (.dodsrc)

The following section refers only to DODS clients from release 3.2 and after.

When a DODS client starts up, it checks to see whether the user has an initialization file available to control the setting of a number of parameters having to do with caching, proxy servers, and other http issues. If this file is not found, one is created in the user's home directory, using default values.[1]

The client initialization file is usually called `.dodsrc`, and is usually located in the user's home directory. You can change this by creating an environment variable called `DODS_CACHE_INIT` and setting it to the full pathname of the configuration file. As of DODS version 3.4, you should use `DODS_CONF` instead. `DODS_CACHE_INIT` is deprecated, and will disappear in future releases.

Here is a sample configuration file:

```
1 # Sample configuration file
2 USE_CACHE=1
3 MAX_CACHE_SIZE=20
4 MAX_CACHED_OBJ=5
5 IGNORE_EXPIRES=0
6 CACHE_ROOT=/home/user/.dods_cache/
7 DEFAULT_EXPIRES=86400
8 PROXY_SERVER=http,http://dcz.dods.org/
9 PROXY_FOR=http://dax.dods.org/.*,http://dods.org/
10 NO_PROXY_FOR=http://dcz.dods.org
11 DEFLATE=1
12 ALWAYS_VALIDATE=0
```

Comments

Starting a line with a `#` makes that line a comment.

Caching

The parameters on lines 2 through 7 control caching. The DODS client can store data you've requested on your local computer. If you repeat a request, the data can be retrieved from this local cache, saving the expense of a network connection. Most web browsers operate the same way. You can control the caching behavior with the following configuration file parameters.

[1] The default values enable a maximum cache size of 20 megabytes, a cache expiration time of 24 hours, and no proxy servers.

CACHE_ROOT This parameter contains the pathname to the cache's top directory. If two or more users want to share a cache, then they must both have read and write permissions to the cache root.

MAX_CACHE_SIZE The value of MAX_CACHE_SIZE sets the maximum size of the cache in megabytes. Once the cache reaches this size, caching more objects will cause cache garbage collection. DODS will first purge the cache of any stale entries and then remove remaining entries starting with those that have the lowest hit count. Garbage collection stops when 90% of the cache has been purged.

MAX_CACHED_OBJ This parameter sets the maximum size of any individual object in the cache in megabytes. Objects received from a server larger than this value will not be cached even if there's room for them without purging other objects.

Many web documents, and DODS data, are delivered with an expiration date in their header information. Generally, this is done for time-sensitive information that may not be valid after the expiration date. You can control the behavior of the DODS client with respect to expiration dates with two configuration file parameters.

IGNORE_EXPIRES If the value of this parameter is 1 (one), then expiration dates will be ignored, and the caching behavior will be ruled by the DEFAULT_EXPIRES parameter.

DEFAULT_EXPIRES Any data received without an expiration time will expire in the number of seconds given by this parameter. If IGNORE_EXPIRES is zero, this will apply to all data, whether or not it comes with an expiration date. The value is given in seconds. The configuration in the example is set for 24 hours.

Proxy Servers

A DODS client can negotiate proxy servers, with help from directions derived from its configuration file. There are three parameters that control proxy behavior. There can be more than one of each of these declarations.

PROXY_SERVER This identifies a proxy server to use for all DODS requests, except for requests specifically modified by the other two proxy behavior directives. The format is:

PROXY_SERVER=*protocol*,*proxy_URL*

Where *protocol* is the name of an internet protocol, and *proxy_URL* must be a full URL to the host running the proxy server. HTTP is the only internet protocol supported by DODS, so *protocol* will always read **http**. There can be more than one proxy declaration, in which case, the DODS client will use the first proxy server on the list that responds.

PROXY_FOR The PROXY_FOR parameter provides a way to specify that URLs which match a regular expression should be accessed using a particular proxy server. The syntax for PROXY˙FOR is:

PROXY_FOR=*regular expression*,*proxy_URL*[,*flags*]

Where *regular expression* is an expression which matches the URL or group of URLs. For example 'http://dax.dods.org/.*ḣdf' would match a URL ending in '.hdf' at dax.dods.org. The regular expression uses the POSIX basic syntax.*proxy_URL* is the same as above.

The *flags* parameter is an optional integer that configures the regular expression matcher. A value of zero sets the default. The four flag values and their meanings are:

REG_EXTENDED If set, use the POSIX extended syntax regular expressions. To set this, add 1 to the value of *flags*.

REG_NEWLINE If set, then . and [^...] don't match newline. Also, the regular expression matcher will try a match beginning after every newline. Set this by adding 2 to *flags*.

REG_ICASE If set, then we consider upper- and lowercase versions of letters to be equivalent when matching. Set by adding 4 to *flags*.

REG_NOSUB If set, then when PREG is passed to regexec, that routine will report only success or failure, and nothing about the registers. Add 8 to *flags* to set this.

You can find a brief tutorial to regular expressions in the DODS bookshelf. See the DODS documentation page at DODS Home page.

NO_PROXY Use this parameter to say that access to a certain host should never go through a proxy without using the more complicated regular expression syntax. The syntax of NO_PROXY is:

NO_PROXY=*protocol*,*hostname*

Where *protocol* is as for PROXY_SERVER, *hostname* is the name of the host, not a url.

Compression

Many DODS servers support compression of the returned data. This can save network bandwidth, and transmission time. You can tell your client to ask the server for compressed data by including a line in your `.dodsrc` file like this:

```
DEFLATE=1
```

Using a value of zero will make the client request only uncompressed data. If the directive is omitted, the default value is zero.

Validation

Caching data locally can be risky if the data in question change often. The `ALWAYS_VALIDATE` option controls whether the DODS client checks the validity of the cached data. The validity check in this case is simply a comparison of the date the data was cached with the "Last-modified" date of the remote data. If the configuration value is set to 1, the client will always validate the cached data. If set to 0, the data will not be validated (but may expire, according to the cache policy set by the `DEFAULT_EXPIRES` configuration directive).

If the directive is omitted, the default value is zero. That is, the default behavior is not to validate the data.

B

Software you will need for DODS

To do anything with DODS, you'll need to be able to unpack the archive files you can download from the DODS site. To save space and transmission time, the archive files are compressed with the `gzip` program. You will have to have a copy of that program to unpack the DODS software.

Most of the software you need for DODS is avaliable from the GNU archives. Refer to http://www.gnu.org for instructions. Look at http://www.gnu.org/order/ftp.html for a list of mirrors of that archive. Use the mirror closest to you, the transmission will be faster.

gzip This is the GNU compression and de-compression program. You will need to install it before you can unpack any of the other software described here. This package is *not* available in the DODS distribution, since it is used to unpack the distribution archive files.

Follow the instructions to install each of the following software packages. Typically, you would install a package called `foo` as follows:

```
gzip -dc foo.tar.gz | tar xvf -
cd foo
./configure
make
make install
```

This is simply a guide, of course, and the installation instructions for each software package should be followed carefully.

B.1 Running a DODS Server

If you use one of the platforms for which DODS supplies a binary distribution, you only need the following software to run a DODS server.

Perl Perl is used for the server dispatch script. (See Section 5.1 on page 50.) This is the main CGI program constituting the DODS server. You must have Perl version 5 or later. (Alternatively, you can also rewrite the dispatch script to use another scripting language, such as your shell. However, we think installing Perl is generally a simpler task.) You can get Perl from the GNU archives, or from http://www.perl.com.

B.2 Running a DODS Client

If you use one of the pre-compiled, out-of-the-box, DODS clients, you will need no additional software to run DODS. However, you can use the "GUI" feature of the DODS client[1] by installing the following software. We recommend this, as it provides useful information about the progress of data transmission or error conditions.

Tcl/Tk The Tcl language and Tk libraries are available from http://www.scriptics.com. You should install the entire package, including the `wish` interpreter program[2] and the `expect` package. The `wish` interpreter is part of the Tcl/Tk core distribution package. This package is also available in the DODS distribution, but the one available from the Tcl site may be more current.

[1] This is not to be confused with the DODS Matlab or IDL GUIs, which are clients of their own. This is simply a client feature that can display transmission and error information to the user.

[2] You can also use a safe Tcl interpreter. Refer to the Tcl documentation for information.

B.3 Building DODS

If you need to build the DODS software, or link it to existing libraries, you will need the following GNU software. Refer to http://www.gnu.org for instructions. Look at http://www.gnu.org/order/ftp.html for a list of mirrors of that archive. Use the mirror closest to you, the transmission will be faster.

GNU C++ Compiler DODS needs `g++`, the GNU C++ compiler to compile.

binutils The GNU linker is part of this package.

libstdc++ The standard C++ library.

GNU Make GNU Make is not essential, but will make like easier.

flex The GNU lexical-analyzer generator

bison The GNU parser generator.

Glossary

alias

A synonym. DODS uses aliases in its attribute-naming scheme. An attribute can have an alias, or second name, by which a user can identify it. An alias can have aliases of its own, but at this point, it becomes difficult to keep track of what points to whom, and we do not recommend this.

Application Program Interface (API)

An API is simply a collection of functions and data types a program uses to access some service. The data types and functions defined in `stdio.h`, such as `printf()`, `fseek()`, `FILE`, and `fputc()`, constitute a commonly used API for C program I/O. The advantage of an API is that it insulates the user from the implentation details of the program.

Array

An array is an ordered set of variables. The members of a DODS array must all be of the same data type. Array members may be accessed with the `[]` operator. That is, `array[4]` specifies the fifth member of `array`. Note that the index of the first array member is zero. Arrays with multiple dimensions are defined as single-dimensional arrays of arrays. For example, a two-dimensional array is an array of arrays.

attribute

A quality of a data variable. This could be the method used to measure the variable's value, the name of the scientist who measured it, the color of the sky at the time, or whatever might be relevant.

Common Gateway Interface (CGI)

A CGI program is a program that is executed by a httpd server upon receiving an appropriately configured URL. The DODS server is a CGI program. CGI is an Internet standard. DODS uses version 1.1.1 of the CGI standard.

compound data type

A compound data type is one that is constructed from other data types. A compound type can be built from simple base types or from other compound data types. *Arrays*, *Lists*, *Grids*, *Sequences*, and *Structures* are all compound types, because they are all defined as aggregates of other data types.

constraint expression

A constraint expression is appended to a DODS URL to select data from the data set identified by the URL.

constructor type

See *Compound Data Type*

container attribute

A container attribute contains other attributes. The analogy is with a compound data variable, such as a *Sequence* or *Structure*, that contains other variables.

daemon

A daemon is simply a process that runs unattended on a UNIX computer. An `httpd` server is generally run as a daemon. It's not clear how the peculiar spelling came into use.

Data Access Protocol (DAP)

The DAP is the method a DODS client uses to retrieve data from a DODS server. The DAP consists of an *intermediate data representation*, an *ancillary data format*, a *procedure* for requesting the data from a server, and an *API* with which to execute the protocol.

Data Attribute Structure (DAS)

This is a DODS construct, showing a list of variables, and attributes associated with those variables, for a given data set. A variable's attributes may include such things as the instrument that recorded it, quality control information and so on. The response to a `_das` request to a DODS server is a DAS.

Data Description Structure (DDS)

This is a DODS construct, showing a textual representation of a data set's data model. The response to a `_dds` request to a DODS server is a DDS.

data dictionary

This is a JGOFS construct. The JGOFS API uses data set names instead of file names to refer to data sets. The API uses the data dictionary to look up the data set names, where it finds the file names or URLs to which the name refers.

data model

The data model of a particular data set can be defined as the set of relationships between the variables that make up that data set. The important thing to remember is that it is this relationship that provides meaning to each of the numbers recorded in that data set. For example, without the relationship of the adjacent location measurement, a temperature measurement is just a number with no meaning.

dataset

A quantity of data, considered as a unit. A dataset may occupy one computer file, or several. A DODS dataset is a dataset that is served through a DODS server.

DODS

Distributed Oceanographic Data System. They wrote a book about it once.

global attribute

A data attribute that applies to an entire dataset.

Grid

The Grid data type consists of an array with named dimensions, and a one dimensional array corresponding to each dimension. It is used to define data grids with irregular spacing.

GUI manager

The DODS core software can create a client-side sub-process with which to manage the user's screen display. Most clients adapted to using DODS will not be able to display intermediate results of a data query, nor will they be able to make sense of network error messages. The GUI manager creates a path whereby messages can travel from the DODS server to the DODS client core software to the user without returning to the client application. The DODS core software can also use the GUI manager by itself, without messages from the server.

HTML

Hyper-text Markup Language. This is the text formatting language in which web pages are written.

httpd

The httpd server is the web server. Web clients, such as browsers like Netscape or Mosaic, send messages to httpd servers on the machine identified in a URL. The return messages from these servers are the data that is displayed to the user.

info service

The `info` service provides information about the usage of a particular server. This is meant to include such information as any functions defined by the server for use in constraint expressions, error messages, the revision of the server software, and a list of any data model translations defined for that server.

lazy evaluation

A method of evaluating a logical expression where evaluation halts after further evaluation could produce no change in the result. For example, when evaluating a string of sub-expressions linked by a logical *AND*, a lazy evaluator would halt after the first false sub-expression, because evaluation of subsequent sub-expressions would not change the result.

List

A List is an unordered list of variables. DODS list members must be of only one data type, but the type may be any of the base or constructor types.

Sequence

A Sequence is a data type similar to a structure, but multi-valued. It is possible to think of a Sequence as an array of Structure values. Unlike an Array, however, only one value of the Sequence, corresponding to the Sequence state is available at any one time.

state

See Sequence. A sequence can be thought of as a multi-values structure. Unlike an array, where all the variable's values are available at once, the values of a sequence's members are only available for the current state. When the state advances, new variables become available.

stride

A stride value is used to select a hyperslab from an array. If `d` is a 10 by 10 array, then `d[0:2:9][0:2:9]` is a hyperslab consisting of every second point in both dimensions.

Structure

The Structure data type is a set of variables, similar to a structure in the language.

Uniform Resource Locator (URL)

A URL is a name that is unique across the Internet. It is analogous to a file name on a single machine in that it identifies some resource that might be data or a program.

Usage Service

See info service.

Acronym Glossary

Here is a list of the acronyms you're likely to run across while reading about or using DODS.

ADDE Abstract Data Distribution Environment (of the SSEC of U.Wisc)

AICC Arctic Icebreaker Coordinating Committee

AIST Advanced Information Systems Technology (ESTO-GSFC) see: http://esto-doc.gsfc.nasa.gov/

API Application Program Interface

ArcIMS Arc Internet Map Server

ASCII American Standard Code for Information Interchange

AXL Arc eXtensible Markup Language

BAA Broad Agency Announcement

BLOB Binary Large OBject

BMRC Bureau of Meteorology Research Centre

BSM (OGC Basic Services Model)

CDC Climate Data Center

CDC Climate Diagnostics Center

CESSE Center for Earth and Space Science Education (Camb., Mass.)

CEOS Committee on Earth Observation Satellites

CGI Common Gateway Interface

CIP Catalog Interoperability Profi le

CIRES Cooperative Institute for Research (in) Environmental Sciences

CLIVAR Climate Variability and Predictability

COARDS Cooperative Ocean-Atmosphere Research Data Standard

COM Common Object Model

CODAR

CORBA Common Object Request Broker Architecture

CORE Consortium on Ocean Research and Education

CSC Coastal Services Center

DAAC Distribute Active Archive Center

DAP Data Access Protocol

DBCP Data Buoy Co-operation Panel

DE Digital Earthhttp://www/digitalearth.gov/

DECEND Deep Submergence Science in the Next Decade
http://www.unols.org/dessc/descend/descend.htm

DESSC Deep Submergence Science Committee

DIAL

DIF Directory Interchange Format

DISC Data and Information Services Center (of GES).

DISS Data and Information System and Services (of Earth System Science and Applications?)

DLESE (Digital Library for Earth System Education)

DNS Domain Name System

DODS Distributed Oceanographic Data System

DRDS DODS Relational Database Server

DSP

ECHO EOSDIS ClearingHOuse

ECMWF

EDG EOS Data Gateway

EDISP Earth Data Information Systems Project

ELIS Environmental Legal Information Systems

EOS Earth Observing System

EOSDIS EOS Data and Information System

EPA Environmental Protection Agency

ESDIS Earth Science Data and Information System Project (GSFC)
http://spsosun.gsfc.nasa.gov/ESDIShome.html

ESE (NASA) (see DLESE?)

ESG (II) Earth System Grid II

ESRI Environmental Systems Research Institute

ESSA Earth System Science and Applications

ESTO Earth Science Technology Offi ce(of GSFC) http://esto.gsfc.nasa.gov

FIC Fleet Improvement Committee

FGDC Federal Geographic Data Committee

FSU Florida State University

GAC Global Area Coverage

GeoTIFF

GCMD Global Change Master Directory

GCRMN Global Coral Reef Monitoring Network

GES GSFC Earth Sciences

GILS Government Information Locator Service

GIS Geographical Information System

GLOSS Global Sea Level Observing System

GML Geography Markup Language

GN Geography Network

GODAE Global Ocean Data Assimilation Experiment

GOOS Global Ocean Observing System http://ioc.unesco.org/goos/whatis01.htm

GOOS IOS GOOS Initial Observing System

GPS Global Positioning System

GrADS Grid Analysis and Display System

GRASS Geographic Resources Analysis Support System

GSFC Goddard Space Flight Center

GSO Graduate School of Oceanography

GTS Global Telecommunications System

GTSPP Global Temperature and Salinity Profi le Programme

GUI Graphical User Interface

G-XML Geography Markup Language

HDF Hierarchical Data Format

HDF-EOS Hierarchical Data Format - EOS

html Hyper Text Markup Language

http the hypertext transport protocol

IDL Interactive Display Language

ITR Information Technology Research(NSF)

JDBC JAVA Database Connectivity

JGOFS Joint Global Ocean Flux Experiment

JPL Jet Propulsion Laboratory

JSP JAVA Server Pages

LAS Live Access Server

LDEO

MARS Meteorological Archive and Retrieval System(Neville Smith)

MATLAB Matrix Laboratory

MBARI Monterey Bay Aquarium Research Institute

MEL Master Environmental Library

MIME Multipurpose Internet Mail Extensions

MIT Massachusetts Institute of Technology

MMS Minerals Management Service

MODIS Moderate Resolution Imaging Spectroradiometer

MPA Master Price Agreement (State of RI)

MrSID Multi-resolution Seamless Image Database

NIAC – NASA Institute for Advanced Concepts

NAML National Association of Marine Laboratories

NASA National Aeronautics and Space Administration

NCDC National Climatic Data Center

NCEP National Center for Environmental Prediction

NetCDF NETwork Common Data Format

NITF

NGDC National Geophysical Data Center

NCDDC National Coastal Data Development Center

NNDC NOAA National Data Centers

NOAA National Oceanic and Atmospheric Administration

NODC National Oceanographic Data Center

NOPP National Oceanographic Partnership Program

NRC National Research Council

NRL Naval Research Laboratory

NSF National Science Foundation

NSIDC National Snow and Ice Data Center

NVODS NOAA Virtual Ocean Data System

ODBC Open Database Connectivity

OGC Open GIS Consortiumhttp://www.opengis.org/

OGCTC OGC Technical Committee

OSU Oregon State University

OOPC Ocean Observation Panel for Climate

OVD Ocean Visual Database

PES Planet Earth Sciencehttp://www.planearthsci.com

PICat Publishable Inventory and Catalog

PMEL Pacifi c Marine Environmental Laboratory

PO-DAAC Physical Oceanography - Distributed Active Archive Center

POV Point of View

RARGOM Regional Association for Research on the Gulf of Maine

RCP Remote Procedure Call

RDB Relational Data Base

RFC Request for Comments (Specifi cation Document). http://www.faqs.org/rfcs/

RFQ

SAF Satellite Application Facility

SAIC Science Applications International Corporation

SCCWRP Southern California Coastal Water Research Project Authority

SCESPS Standing Committee on Earth Science Products and Services

SDE Spatial Database Engine

SDK Software Development Kit

SDTS

SGT Science Graphics Toolkit (PMEL)

SIG

SOAP Simple Object Access Protocolhttp://www.w3.org/TR/SOAP/

SOOP Ship of Opportunity Program (GOOS)

SSEC Space Science and Engineering Center (U. Wisconson)

SST sea surface temperature

STT Space Time Toolkit (UAH)

SVF Single Variable Format

TAO Tropical Atmosphere Ocean array

TEDS Tactical Environmental Decision Support System

TCP IP Transmission Control Protocol/Internet Protocol

THREDDS THematic Real-time Environmental Distributed Data Services

UAH

UCAR University Corporation for Atmospheric Research

UFS Universal File System

UMAC Upper Midwest Aerospace Consortium

UNEP United Nations Environmental Program

UNOLS University-National Oceanographic Laboratory System

URI University of Rhode Island

URI Uniform Resource Identifier (See RFC 2396)
http://www.faqs.org/rfcs/rfc2396.html

URL Uniform Resource Locator

USGS United States Geological Survey

VOS Voluntary Observing Ship

VPF Vector Product Format

WCS WebCoverage Server (OGC BSM)

WFS Web Feature Server (OGC BSM)

WHOI Woods Hole Oceanographic Institution

WMS Web Mapping Services OGC WMS (Web Mapping Services) spec

WRS Web Registry Server (OGC BSM)

WMT Web Mapping Testbed. See WMS.

WOCE World Ocean Circulation Experiment

WWW World Wide Web

XML Extensible Markup Language

Bibliography

[1] The Free Software Foundation, Cambridge, MA. *GNU Emacs*. Version 19.

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

[3] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Ma., 1994.

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

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

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

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W