DODS—Data Delivery Architecture

23 August 1996

Abstract

This paper describes the architecture of the DODS data delivery mechanism. This mechanism is one of two components that make up the DODS system design, the other being the data locator. In this paper it is assumed that every data set is accessible using an application programmer interface (API). In addition to local data set access, the DODS data delivery architecture provides remote access by replacing existing API implementations (which provide only local access) with new implementations which access data through servers located on the Internet. For each API DODS supports, a client library and data server are created. The client library supports the same functions as the original implementation of the API, but uses a DODS data server to satisfy each function call rather than reading data from a file. The client library and data server communicate using an application level transmission protocol that facilitates translation between the supported APIs. While the data delivery architecture of DODS is not inherently limited to the UNIX operating system, the DODS project has no plan to support other operating systems. It is also important to note that this architecture describes a method for direct access to scientific data.

Note

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

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

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

Contents

1 Introduction

This document is an introduction to the DODS data delivery system. It presents an architectural view of the system and provides rationale for some of the important decisions that lead up to this design. This document is not a detailed description of the design; that can be found in DODS—Data Delivery Design[1] and related documents.

The data delivery system uses the client-server paradigm to provide access to remote data sets. On each computer which hosts a data set, a data server must be installed. Client programs which communicate with these servers can be built either from scratch or from existing code using reimplementations of commonly used data access Application Programmer Interfaces (API)[2]. The API libraries are reimplemented so that existing code can be re-linked *without* modification. Thus the data delivery system provides a pathway for existing code to migrate from local access to remote access without requiring modification of that code.

DODS addresses access to on-line data only. It would be nice to provide access to data that are not already on-line, but the problems associated with that are outside the scope of DODS (e.g., some data collections are not suited to on-line access because of their size or content). Thus, the data delivery architecture addresses the problem of network access to data sets that are already on-line.

Before describing the data delivery system, some important computer networking and data management terms and concepts are discussed. Each of these is first defined in general terms and then particular aspects of those ideas that are important to DODS are discussed. These include networking technologies (remote procedure calls and the hypertext transfer protocol) and data access tools and concepts (application programmer interface, data model and data format).

2 Important Terms and Technologies

Some important concepts appear frequently in the discussions about the DODS data delivery system. In this section they are defined or described so that readers will all have some common background when reading the documents that are part of the data delivery system.

2.1 Application Programming Interface—API

An application program interface (API) is a formal method of access to a computational resource such as a disk file, set of disk files or a device such as the display screen.

The DODS data delivery architecture addresses access to data stored on-line on computer systems. In this paper we assume that an API is used to access data

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

[2] As of 23 August 1996, DODS supports two APIs: NetCDF and JGOFS.

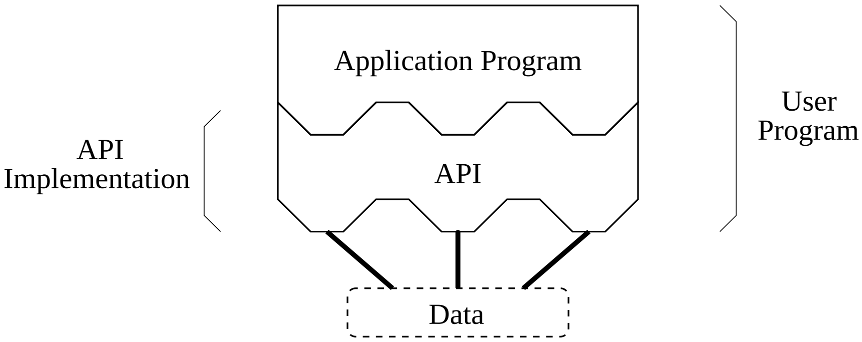

Figure 1: Architecture of a Simple User Program—An API is Used to Access Data in a Disk File.

stored on a random access storage device such as a disk drive. However, it may be the case that some data are not actually accessed via disk files but through a more complex mechanism such as specialized database hardware systems. None-the-less, the architecture of the data delivery design is still based on the *concept* of access to data through an API, stored in disk files.

A specific API implementation consists of a set of operations used to access an instance of a well defined resource. The set of operations are typically functions such as open, read, etc., that the system uses to provide access to the resource. API operations are made available to the application program usually as function calls in a software library. Figure 1 shows a user program linked with an API that is used to access some data.

As is shown in Figure 1, an API consists of an external interface and some internal implementation. The external interface is used by the application program. This external interface is assumed to be very constant throughout the life of the API. User programs can expect that future implementations of the API may improve performance or provide additional features, but will still present the original set of function calls, possibly with some additions over time.

By basing data access on existing APIs, DODS is reusing software systems for which stability and longevity were principle design goals.

2.2 Remote Procedure Call—RPC

Remote procedure calls (RPCs) provide a framework for implementing remote access to a system. They create a distributed computing environment that is established and controlled at the procedure level within an application.

In an earlier version of DODS, RPCs were the basis for all data transmission; the data delivery design has since changed to HTTP as the foundation for data transmission. Regardless, RPCs still merit discussion since they frame the concepts that are central to the design of the data delivery component of DODS.

RPC technology simplifies the design of distributed software systems by providing a means to separate one program into two cooperating processes using a procedural interface. This is straightforward because many programs can naturally be broken into two sections along a procedural line separating a core set of functions and some additional functions special to a particular application. Figure 2 shows how RPCs can be used to split a user program and API into a client program and remote server. The RPC client and server 'stubs' now form the communication interface between the application program and server. The client stubs encode their arguments and sends them over the network to matching stubs in the server. The server stubs extract the arguments from the network and compute results based on the arguments. The server stubs return to the client the results of the computations.

RPC client stubs can serve as replacements for the API functions invoked by an application program. In this case the function of the networking portions of the client and server software are no different than in the more general case described in the preceding paragraph. Once arguments are passed into the server stub it can 'compute the result' by calling the API function for which the client stub is a replacement. The server stub then returns as its result the return value of the API call. From the vantage point of the user program, there is no difference between this access and an access to a local file using the standard API library. This is true because the RPC client stubs preserve exactly the semantics of the standard API since there is a one-to-one mapping between function calls in the RPC client stubs (i.e., client library), the RPC server stubs (the transmission protocol) and the standard implementation of the API[3]. While building a server in this way is guaranteed not to change the semantics of the API, the operational mode of the API has radically changed. Previously it read data from a file, now it reads data from a data server which may be located anywhere on a given computer network (e.g., the Internet).

RPCs are not without drawbacks, however. A major problem with RPC technology is that it is based on a strict request-reply paradigm. While certain data sets work well in that context, others do not. For example, JGOFSprovides access to data sets that are functionally relational databases, and thus are of unknown length. A good data server for such a data set would begin returning records to the user program *before* the server completes sending the result of a given access. However, the request-reply nature of RPCs makes this difficult.

Further complicating the use of RPCs is that all network interprocess communications code is nominally contained in the RPC server. Thus the server is responsible for all access logging and all security precautions. While basic access to a network is fairly easy to provide, more advanced functions quickly add to the complexity of the server and can easily dominate the time required to design and build it as well as the effort required to support it.

Note

During the early part of the DODS design effort, we built two sample RPC client-stub/server-stub pairs: one for NetCDF and one for JGOFS. While these

[3]In effect, RPC is used to replace the normal procedure call mechanism with network messages, but outside of the argument processing there is no change to the API entry points so their semantics are guaranteed to remain unchanged.

worked well within the limitations of RPCs, it was felt that for certain data sets, particularly those that contained relational data, the strict request-reply nature of RPCs would need to be improved upon. This combined with the additional work required to support development of full-fledged data servers caused us to look at other transport mechanisms.

These implementations, however, did effectively demonstrate that we could build surrogates for the DODS supported APIs and re-link third party programs with surrogate libraries. The re-linked user programs could indeed access data via the data servers without any modification whatsoever. These RPC-based libraries, however, could not interoperate.

2.3 HyperText Transfer Protocol—HTTP

A second data delivery mechanism that is, on the surface, very different from RPCs is the HyperText Transfer Protocol[4][5]. This transport mechanism is primarily intended to move hypertext documents (text, images, ...) over a network. Its basic unit of transfer is a file. The protocol is closely coupled with existing HTTP servers from World Widw Web Consortium, National Center for Supercomputing Applications and commercial providers. These servers support an extension mechanism called the Common Gateway Interface[6] which can be used to transform information on the server computer to a hypertext document. Once the hypertext document has been prepared, it is passed back to the HTTP server which then sends it to the client.

A crucial difference between RPC technology and HTTP is that RPC is used to build an application-level communication protocol while HTTP *is* an application level protocol. While RPC can be used to build either stateless of stateful distributed systems, HTTP can only be used to build stateless systems. Because HTTP's extensions typically take the form of filter programs combined (using the CGI mechanism) with a server, HTTP is a poor choice for supporting 'fat' interfaces. In other words, HTTP/CGI is a good choice for interfaces that are stateless and have a small number of entry points, but a poor choice for stateful or large interfaces.

The widespread use of HTTP and the competition between various providers of HTTP daemons has focused the attention of those server providers on important considerations like system security and accounting. Thus, while providers of HTTP software need to make high quality implementations of the HTTP protocol available, there is also considerable effort expended towards additional features which improve the overall function of any networked system. Because using RPCs virtually necessitates rebuilding support features like access control and logging, using HTTP can translate into significant savings during the design, implementation and management phases of a project.

[4] http://www.w3.org/hypertext/WWW/Protocols/HTTP/HTTP2.html
[5] HTTP is the communication protocol used for the World Wide Web.
[6] http://hoohoo.ncsa.uiuc.edu/cgi/overview.html

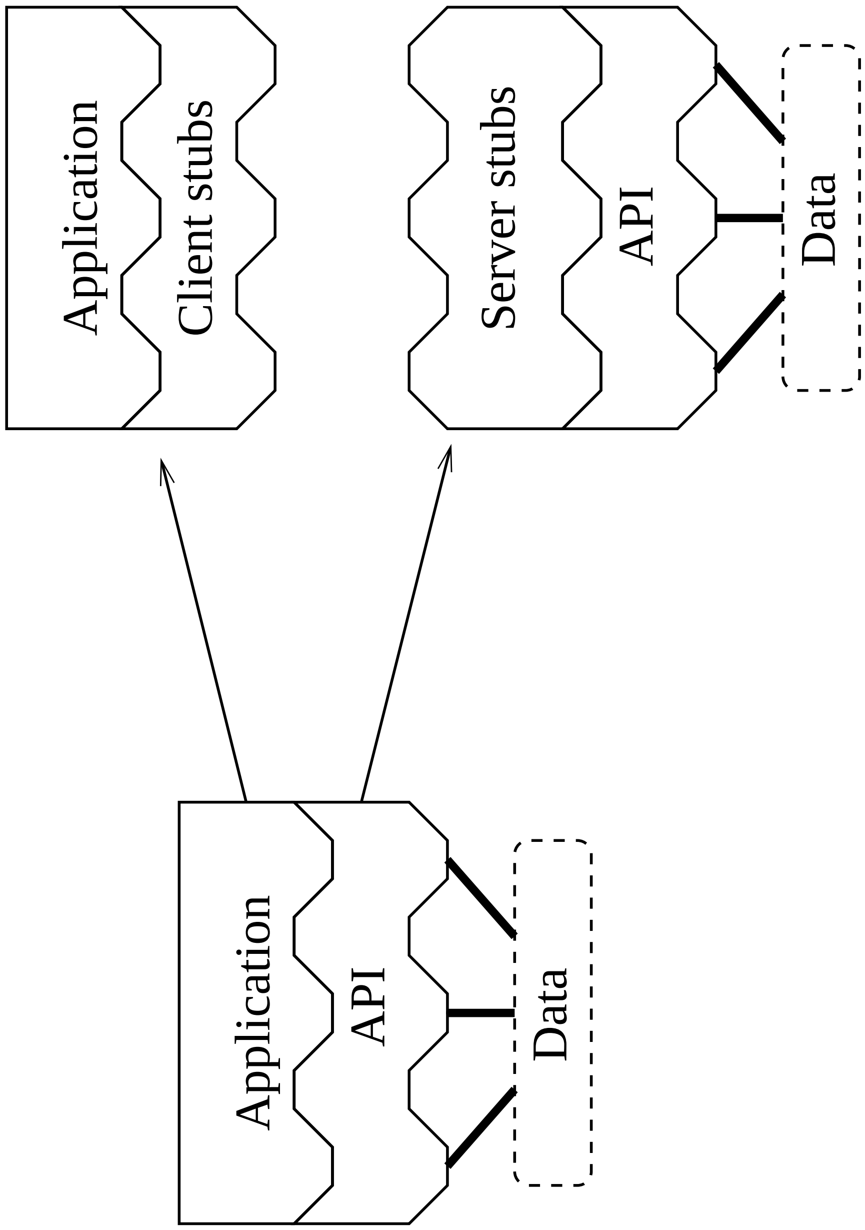

Figure 2: Implementation of Distributed Application—API Access Using RPCs.

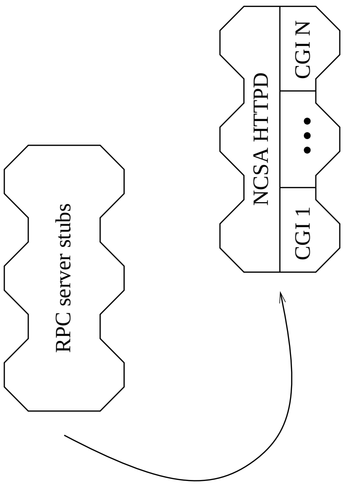

Figure 3: HTTP can be used to construct servers architecturally similar to those built with RPCs. (Figure 2). To do so the *RPC server stub* portion of the server is implemented using HTTPD and one or more *Common Gateway Interfaces* as shown in this figure.

The widespread use of HTTP means that server programs for HTTP are likely to exist on many of the would-be data-provider machines. Thus using HTTP will remove one impediment from a user becoming a data provider, that is, to become a data provider users will not necessarily have to install a data server. The nature of RPC technology is such that data server installation is typically considered complex by users and removing that hurdle improves the accessibility of the system (since the system depends on, to a large extent, the willingness of people to become data providers).

While there are fundamental differences between RPC and HTTP, Figure 3 shows how the *RPC server stubs* layer in Figure 2 can be constructed using one implementation of HTTP and some additional software. One crucial difference between systems built with RPC and those built using HTTP is that the RPC-based server can maintain state information while the HTTP-based server cannot. Thus, while servers based on RPC can use any convenient set of functions as the server interface, the HTTP interface must be more carefully chosen so as to insure that the system can function efficiently without state information retained in the server.

2.4 Data Model

A data model describes the organization of data. It is composed of a defined set of elements and structures along with valid operations which can be performed on those elements and structures. All programming languages incorporate a data model in their design. For example the C programming language defines a set of types which can be used to build programs (int, double, ...); that set of types and their associated operations is the data model for C programs. Similarly, while not full-fledged programming languages, APIs define a data model for the resources to which they provide access by defining a set of functions (which are analogous to operators in a programming language) and defining types which are passed to, or returned from, those functions.

Data models are important because they provide a way to describe the types of information that can (or cannot) be described using a given programming language or API. Thus a data model can be used to compare the applicability of two such systems to a particular problem. In addition, if a single data model is supported by two APIs then it is possible to translate information represented using that data model from one API to an equivalent representation in the other API. Knowing which data model or models are supported by a set of APIs enables a scientist to choose the API best suited to a given type of information. It also provides a means for programmers to build translators for certain types of data represented using those APIs.

As is often the case with semantic descriptions, a formal description of a data model is hard to produce for many existing systems. This is true because many APIs do not rigorously define their operators. Instead they rely on conventions with ill specified boundary conditions. Thus the exact semantics of many data types are hard to formalize.

The lack of formal data model semantics is further complicated when new types are added (ad hoc) to an API. The 'data type' exists only in so much as users of the API agree to be bound by convention. A user may abuse the convention or they may ignore, out of ignorance, a widely used convention. Either case creates data that is hard to use. This problem create special difficulties for people writing data translators since important conventions have not been followed. For this reason, it is impossible to write perfect translators even for a subset of data types.

Lack of formal data model semantics makes it hard to build translators to or from an API or programming language even for a small set of well defined objects. Thus for many APIs, while it is possible to build translators for data represented using a core set of data models common to those APIs, it is not in general possible to build translators for any arbitrary object represented in one API to every other API[7].

2.5 Data Format

It is important to distinguish a data format from a data model. The later is the convention used to store information so that a program can access the information. A data format defines only the form of the information, not the operations that can be performed on it. In contrast, a data model often does *not* define a particular format. For example, an integer data type (which includes properties of the object like its range and the operations which can be applied to it) does not specify exactly how such an object will be stored in a computer. It is possible to build (in software) many different implementations of a given data model which have a high degree of

[7]Furthermore, some data models cannot be translated into other data types without solving a computational problem that is known to be NP complete.

interoperability but which all use different internal representations.

Many data formats imply a particular data model or set of data models. Thus it is possible to translate information stored in one format to another when the two formats share data models. However, the problems that plague many APIs (lack of formalism, ...) are worse for data formats because the set of operations, rather than being at least partially defined, is purely conventional. Thus features of a particular format which may be in fact consequences of a particular implementation often must be accounted for in the definitions of the operations. Lossless translations between data formats are very hard even in limited cases and, as with APIs, impossible in general.

3 Data Delivery Architecture

DODS uses the client-sever paradigm to provide access to data across the Internet. Each computer with on-line data can provide access to any user with network access by installing one or more DODS data servers. Users then employ client programs to access data from the data servers. The data delivery architecture for the entire distributed data system is the set of DODS data servers and clients in existence at any given time. Because there is no central authority which controls clients or servers, the system is in a constant state of flux. In the future there will be a system which enables users to search for servers with a particular type of data, but participation in such a central registry is voluntary and is not part of the data delivery architecture proper, rather the data delivery architecture consists solely of the data servers and necessary client software.

The data delivery architecture does not rely on a centralized archive of data, instead various data providers make relatively small amounts of data available using their computers and data server software. Each user of data can access the sum total of data from all the people or groups making data available much as each user of a World Wide Web browser can access any document made available through HTTP servers. While the contribution from each individual is small, the total pool of data that users can access is potentially huge. Furthermore, the architecture of the DODS data delivery mechanism is both general enough and simple enough to be used by individual scientists and large agencies or groups of scientists. Thus the pool of data which comprise DODS can be contributed by a wide variety of providers.

In the data delivery mechanism, the DODS data access protocol is used to provide the interface definition for a data server. However, each data server translates the 'calls' in the protocol to function calls in the API. These calls are used to encode information contained in the data set. Thus, DODS data access is built around a family of data servers, defined by the set of data-access APIs supported by DODS, all of which use the same data access protocol[8].

[8]In previous versions of this and other documents, the data access protocolwas referred to as the DODS API; that named seemed to confuse more than edify, so it has been changed.

An essential concept on which DODS bases its data delivery architecture is that a significant percentage of existing data is stored in a documented format or API[9]. This assumption ensures that there is a high return for time spent developing data servers. If an API has been used to encode a large number of different data sets, then a single type of data server (one for that API) can be used to provide access to all those data sets. The assumption that the interface to a data set (or type of data set) is well documented also reduces the complexity of designing a data server.

A second essential concept is that many science-data APIs draw on a small set of data models. These data models include relational sets, arrays, hierarchies, etc. It should therefore be possible to construct data servers which can translate some (or many) data sets from one API to another. This will enable users of one of the DODS supported APIs to read most of the data sets in any of the other supported APIs—not just data sets in one API.

The DODS data delivery architecture is flexible enough to support a wide range of users. Some users will have a detailed understanding of the format and content of a particular data set and should be able to use that understanding to their advantage. Other users will have no prior knowledge of format or content of the same data set and still should be able to access that data in a meaningful way.

Supporting a wide range of data access, from data browsing to direct access, means providing many different types of user applications which can access data. In fact, the sum total of data browsing, visualization and access software that even a single user might want is staggering. Furthermore, much of that software has already been written, but typically not with distributed access in mind and not for the specific format or API in which the data a given user wants is encoded. Complicating the problem is the wide range of data formats and APIs now available and the rate at which their numbers are expanding. Trying to write a complete set of network tools would be impossible. Before the tools were complete new formats/APIs would need to be supported.

A solution to this problem that fits well with the idea of data servers providing access based on APIs is to build replacement libraries for for those APIs. These libraries present to a program the same set of functions that the original implementation of the API presented, but replace the bodies of those functions with code that access a data server rather than a local resource. Any user program that was originally designed to work with one of the supported APIs can be re-linked with the DODS implementation of that API (called a 'surrogate library' or a 'client library') and access all of the distributed resources of DODS. Thus, the parts of the data delivery architecture that support data browsing, visualization and access are not described here because DODS will rely on existing user programs to provide those functions.

The DODS architecture supports data management by presenting the scientist with a set of third-party tools for storing and organizing data. Each of these tools

[9]In the remainder of this paper (and elsewhere) we use the phrase 'data API' to mean the API used to encode the data. From time to time, we abuse the phrase 'data API' to mean 'the format used to store the data'.

provides support for representing information using one or more data models. Furthermore, the inclusion of an API in DODS implies that the API has been accepted by the scientific/oceanographic community and is thus of use in organizing or accessing relevant data sets (i.e., we chose APIs which are 'popular' and feel that popularity implies acceptance). The set of data models circumscribed by the set of supported APIs forms the set of data models which the data access protocol uses to describe and provide access to data sets. Because the set of models is finite, it is possible that the user community will want to add an API which uses a data model DODS knows nothing about. In this case the data access protocol must be extended to support the new data model(s).

The complete data delivery architecture for DODS consists of data servers and data sets located on computers throughout the Internet, combined with user programs re-linked with DODS surrogate libraries. The set of data sets and user programs are not limited to those that use DODS supported APIs and/or formats. That a data set is accessible via a DODS data server and that a user program is able to access data using the data access protocol is the sole criteria for membership in DODS.

The intent of this architecture is to make it as simple as possible for scientists to access remote data. Using DODS servers, it should be possible for many scientists to use programs already written to access new and existing data sets stored on remote computers. The only preconditions their programs must meet are that they must have been written using an API which DODS supports and that they can be re-linked with the DODS implementation of that API. Because the interface to the API remains exactly the same the user program does not need to be modified in any way to access the remote data available through DODS.

3.1 Two Prospective Models for Data Servers

Typically, a user program can be decomposed into some user-written source code and a library of functions which implement the data access API. This library is typically written and maintained by some third party group (e.g., a commercial software vendor or a research organization). Users of a particular API access DODS data servers by replacing the 'standard' version of that API with the DODS version. This new version presents the application with the same functions (so that all their existing programs continue to work just as before), but adds the capability to access remote data sets by connecting to DODS data servers. The server and the user program/client-library combination communicate over the Internet and thus may reside on any two machines that are connected to the Internet. Each client library for a particular API is built so that a user program which was written for that API, with or without prior knowledge of DODS, will be able to access any of the data available through DODS servers. The DODS client library for a particular API will necessarily perform different operations internally, but the interface it presents to the user program will be exactly the same as the original implementation of the library.

There are two conceptual models for a DODS data server: one in which data is accessed using a server which closely mirrors the function calls supported by the API used to store that data, and one which supports an interface using a different API. In the latter case, calls in the server's interface (i.e., transmission protocol) are translated into calls in the data API and the results are translated back from the data API to the server interface for transmission. Figures 4 and 5 show these two architectures.

If DODS were to only support a single API, then clearly the first model of data server (Figure 4) would be preferable since, lacking the need to transform data representations, it is simpler to implement. A set of client stubs for the API would serve as the reimplementation of the library. User programs would link to this and the internals of each function call would be replaced with code to marshal the function's arguments and send them to the correct server stub. On the server side, the corresponding stub would unmarshal the arguments, call the function as originally implemented and arrange to return to the client the function's result (i.e., the reimplementation of the library, linked with the user program). Once constructed, simple optimizations could be incorporated to such a system to improve network use.

However, DODS intends to support data access through many APIs since one goal of DODS is to provide a means for different data sets to interoperate. One possible server architecture would require DODS to provide software to translate information in each of its supported APIs to each of the other N supported APIs. If each data server provides access solely in terms of the API used to encode its data, then each new API brought into the set of supported APIs requires $N+1$ additional translators be constructed. In effect, the set of APIs becomes closely coupled in this model, that is each new API effects the necessary support software for *all* APIs and future changes in the system require modifications to existing components. As the number of supported APIs increase the work required to add an additional API becomes intractable.

A second server architecture uses a common transmission protocol for all data servers and requires that each supported API be accessible using this protocol. In this model, support for a new API does not require modification or addition to the existing software for currently supported APIs. However, the trade-off necessary to reduce the growth of overall system complexity translates to increased cost to support each individual API because for each API two translation modules must be built, one to go from the API functions to the transmission protocol, and one to reverse that operation. Of course, if the intent is to only provide access to DODS data servers using the new API but, not to build servers for that API's data, then only one translation unit need be built. However, in general both translators will be desirable.

A third architecture involves combining elements of the the preceding two architectures; using servers which speak an API dependent protocol combined with translators that provide access to those servers through a single uniform interface. This design has one advantage over a design that builds translation into the client library and the server—it provides a simple way for an API to be partially added to DODS. However, in many ways this may not be a good feature for the system. If APIs can be added without translators this could significantly limit the functionality of the system. DODS would 'support' many APIs, but only provide translators for a small set of APIs. Even though the set of APIs could grow, the set of data available to most of the APIs would be limited to data sets created using that API.

To control the complexity of adding support for an API to DODS, the second architecture was chosen. Each API is accessed using a single transmission protocol (see DODS—Data Access Protocol[10]). This protocol (which in the past was referred to as the DODS API) is sufficiently flexible to represent information encoded with either of the two currently supported APIs. The translation to and from the user API and data API are performed by the client library and data server, respectively. This design was chosen over a design with data servers that use the data API as a transmission protocol because it was felt that interoperability of all data sets was a central goal of DODS[11].

The client and server components shown in Figure 5 indicate that to add an additional API to the set of APIs supported by DODS, four modules are involved: the surrogate user API library, Client Network Interface, Network Server Interface and Server Side Interface. Of these four modules two, the client interface and the server interface, are unlikely to change from implementation to implementation. Thus, each additional API will involve the construction of the surrogate client library and the server side translator.

Note

> In previous documentation, the *DODS Server* architecture was referred to as a *Translating Server*. In effect all DODS data servers are now translating servers. It is important to understand that a DODS server does not translate the data—it translates the method of access to data.

3.2 The Role of Ancillary Data in Translation

In order to support different data access APIs, DODS must effectively translate between data models (See Section 2.4 and the DODS—Data Access Protocol[13]sections titled "Dataset Descriptor Structure" and "Dataset Attribute Structure"). It is often the case that the information about a data set available from an API is not sufficient to translate that data set's access to another API which implements a different data model. What is needed is some additional information about the organization and relationships of the variables within the data set. This information is contained in the *Ancillary Data* store of a data set (Figure 5).

The ancillary data for a given data set is stored in text files and is completely separate from the data set itself. A user who wishes to provide data to the DODS community can store the extra information in one of these text files without modifying the original data set.

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

[11] As is described in DODS—Data Delivery Design[12], a second benefit of this architecture is that certain transmission optimizations with relational data sets are possible.

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

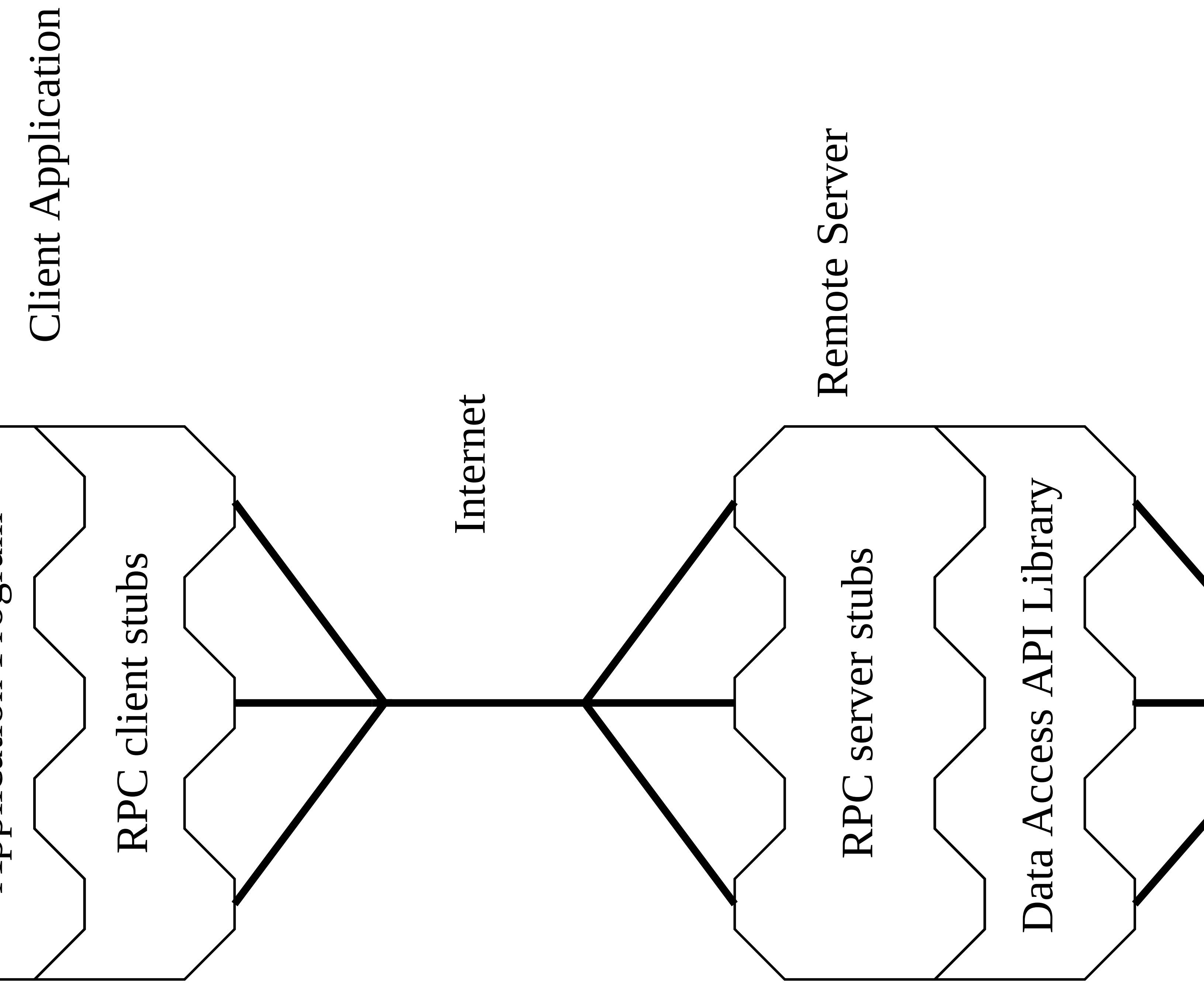

Figure 4: Data server using the API interface as a Network Protocol (via RPC technology)

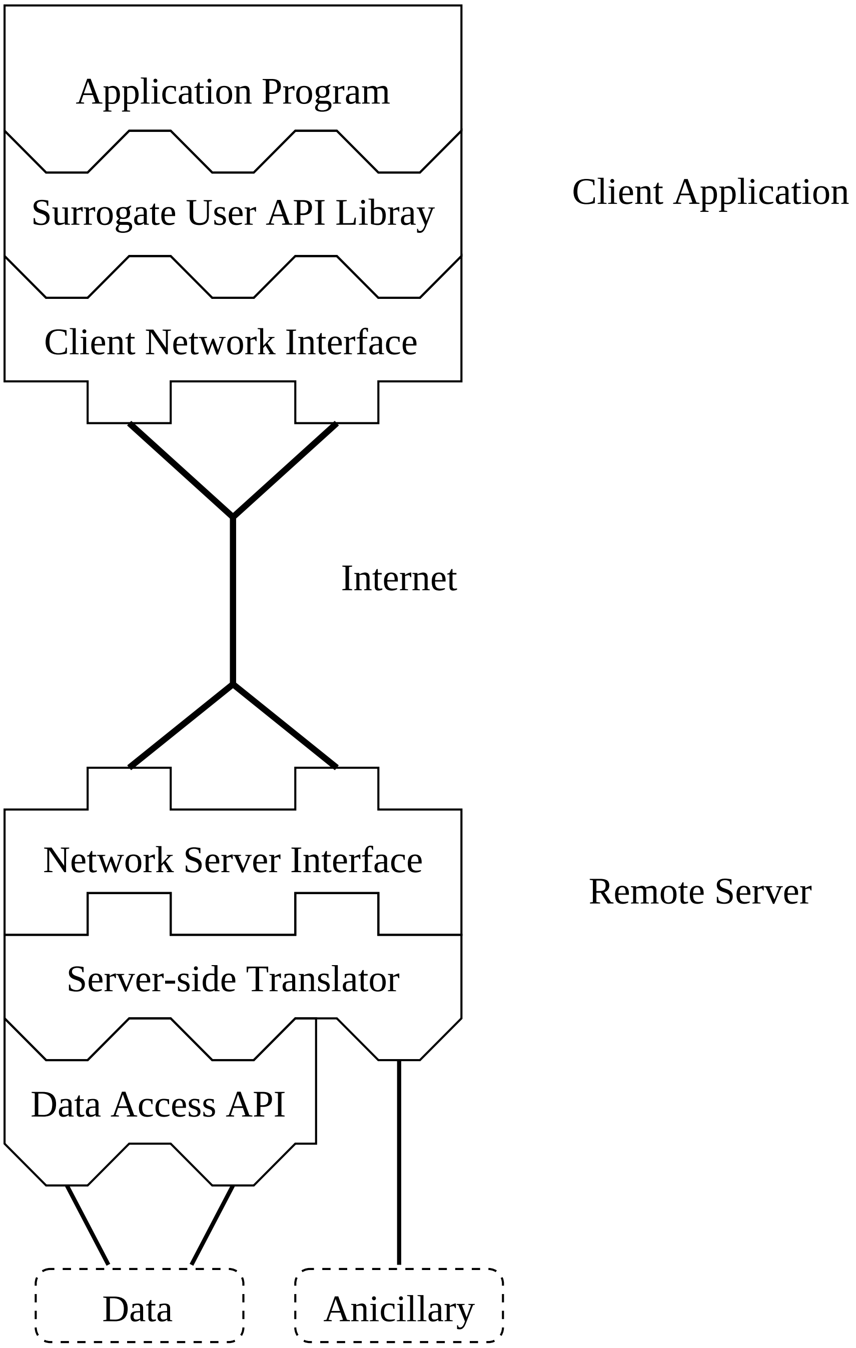

Figure 5: DODS data server configuration employing client and server translators. The network protocol is not the same as the API interface; it is a special transmission protocol which facilitates the translation of calls made in one API to calls made in another.

There are two types of ancillary data supported by the data access protocol. Information about the data types of the variables contained in a data set may be provided using the Dataset Descriptor Structure (DDS). This structure is used to describe the name, type, shape, and structural relation of the variables in a given data set. Whenever the client library needs to translate information from its network form (i.e., the form the information takes when it is encoded using the data access protocol) it will use information in the DDS structure to decide how to perform those transformations. Similarly, the Dataset Attribute Structure can be used to affect the translation of variable attributes in a given data set.

The ancillary data files contain knowledge about the variables for use by the data server during translation. Data set translation is effectively the translation of each variable in a given data set and API combination to one or more variable(s) in a different API. The translation of each variable is a transformation of data type and that transformation is directed by the knowledge the translating software has about the API and the specific types and organization of a given data set. For a data server, the ancillary data structures are the only source of knowledge available beyond the API and the data set itself.

When the information reaches a client library, all content specific to a given API is lost. All information from the data set that is not actual data is coded in a structure that is the catenation of the ancillary data structures and information that can be extracted from the data set itself. This information is structured just like the ancillary data files stored on the server side. These files are the only source of knowledge about the data set the client library can access.

Ancillary information is a data type description language. Knowledge about the data set not available to the data server must be stored in one of the two ancillary data files if it is to be available to the server upon translation. The data server combines information evident from the data set with that in the ancillary files to create the two structures sent to the server. These two structures have the same form as an ancillary data.

4 Conclusion

The DODS data delivery architecture is based on a set of loosely cooperating data servers spread throughout host computers on the Internet. The extent of these host's commitment to DODS is that they run the servers and permit people with client software to connect and access data. This system architecture follows in the footsteps of other client-server systems where participation is wholly voluntary.

To provide a basic level of server security and accounting, the data delivery architecture prescribes that third-party HTTP servers be the principle network interface of DODS. Using these servers as the basis for the data servers frees the DODS project from the task of building secondary support services into DODS serves. The extra work required to build servers robust enough to be considered trustworthy is a considerable drawback to RPC technology.

The DODS data delivery architecture is based heavily on data access through a set of known, stable and documented APIs. Each of these APIs defines, to a greater or lesser extent, the semantics of the data models which it uses and thus the data it can be used to store/access. Given this and that most science data APIs draw on a small set data models, it is reasonable to assume that calls in these APIs can be translated into a common access protocol. Since each client program will use this common protocol to access data servers, DODS will facilitate interoperability between the set of APIs it supports. Because the characteristics of APIs that make this possible are not limited to a small set of APIs it is reasonable to assume that the set of APIs supported by DODS can be extended without radical modifications to the basic structure of the system.

In order to maximize the body of existing software that can be used with DODS, the development of the data delivery system will concentrate on building data servers and reimplementing API libraries. The data servers can be used to provide access to data stored in any one of the supported APIs data sets. The reimplemented libraries (i.e., client libraries) can be used to re-link existing application programs providing the extended functionality of DODS without necessitating any modification of those programs.