To the Graduate Council:

I am submitting herewith a thesis written by Rick Phillips entitled "A NetSolve Client for Windows NT/95." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Jack Dongarra, Major Professor

We have read this thesis
and recommend its acceptance:

Michael W. Berry

Accepted for the Council:

Associate Vice Chancellor and
Dean of The Graduate School

A NetSolve Client for Windows NT/95

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Rick Phillips

December, 1998

Acknowledgments

I want to thank Dr. Jack Dongarra for his patience and support throughout the course of my thesis. I would also like to thank Dr. Michael Berry and Dr. James Plank for their support and service on my committee. I would especially like to thank Dr. Henri Casanova (the primary developer of NetSolve) for his patience, support, guidance, and source code!

I would like to thank David R. Brown for his guidance, support, and encouragement. His kindness and friendship have helped me though some very difficult times. He has been a source of inspiration on many levels. Likewise, I would like to thank Ron Parr for his support, encouragement, and friendship. As my boss, Ron gave me the flexibility and support I needed to complete this work while still managing to pay the bills. As my friend, Ron encouraged me to complete this work, even when it might have been in his best interest as my boss to encourage me to work on other tasks.

I would also like to thank Rob Tannert for his friendship and support. His unique sense of humor and quick wit have lifted my spirits on numerous occasions. His ability to consume massive amounts of salsa is legendary.

I would like to thank Lisa McDaniel for her understanding, emotional support, and companionship. I'd also like to thank Cathey Love and Cathy Higdon for "caring". I'd like to thank Charles Reeves, Karen Fowlkes, Nancy Getsi, Schree Liafsha, Keith Shultz, and Christie Dowling for their friendship.

I would like to offer special thanks to my mother, Ruth Phillips. Her unconditional love and support gave me the confidence and the means to develop into the person I am today.

Finally, I'd like to dedicate this thesis to the memory of my father, Harry M. Phillips (3/29/33 – 6/4/98). I miss you Dad.

Abstract

Software libraries are often used to solve scientific computing problems. There are, however, barriers that limit the usefulness of these libraries. The NetSolve system was developed to remove some of these barriers by allowing users to access software libraries over a network. However, the NetSolve system is not available on all platforms. In this thesis, a NetSolve for Windows client was developed to extend the availability of the NetSolve system to the popular Windows NT and Windows 95 operating systems. The source code for the existing UNIX NetSolve client was used as a starting point and updated to work in the Windows environment. In addition, an extended XDR implementation was adapted for use in the Windows environment. This XDR implementation, as well as the techniques used to update the UNIX NetSolve client source code, provide useful examples and tools for future developers who may need to port UNIX networking software to the Windows environment.

Contents

1 Introduction 1

1.1 Motivation 1

1.2 Objectives 2

1.3 The NetSolve System 2

1.4 Related Work 3

1.4.1 Previous NetSolve Clients 3

1.4.2 PVM Implementation for Windows 4

2 Design Considerations 5

2.1 Overview 5

2.2 Environment Variables, the Registry, and Initialization Files 6

2.3 Sockets 7

2.3.1 The SOCKET Data Type 7

2.3.2 The Select() Function 8

2.3.3 Error Codes 9

2.3.4 The WSAStartup() Function 10

2.3.5 The gethostbyaddr() Function 10

2.4 XDR 12

2.5 Text Files 13

2.6 Development Environment and Tools 13

3 Implementation Issues 14

3.1 Overview 14

3.2 Development Environment and Tools 15

3.3 XDR 17

3.4 C Interface 18

3.4.1 Implementing an Initialization File 19

3.4.2 Implementing Windows Sockets 19

3.4.3 Implementing Static and Dynamic Link Libraries 20

3.5 MATLAB Interface 22

4 Using NetSolve For Windows 24

4.1 Overview 24

4.2 Downloading and Installing the Software 24

4.3 Using the Software 25

4.3.1 General Information 25

4.3.2 Creating Netsolve.ini 27

4.3.3 Using Dynamic Link Libraries 27

4.3.5 Using UPF Files 29

5 Conclusions and Future Work 30

Bibliography 31

Appendices 33

Appendix A: Source Code for DLL Routines 34

Appendix B: Source Code for win_util.c 35

Appendix C: Source Code for communicator.c 41

Appendix D: Source Code for clientutil.c 55

Appendix E: Source Code for socketutil.c 70

Vita 72

List of Figures

Figure 1: The NetSolve System. (a) A client sends a problem to an agent and receives a sorted list of servers that can solve the problem. (b) The client sends the problem to the first server in the list and receives the solution.....3

Figure 2: Primitives in the Berkeley Sockets API.....8

Figure 3: The SOCKET Data Type and INVALID_SOCKET Constant. On the left is a typical Berkeley Sockets source code fragment. On the right is the Windows Sockets equivalent.....9

Figure 4: The SOCKET_ERROR constant and WSAGetLastError() function. On the left is a typical Berkeley Sockets source code fragment. On the right is the Windows Sockets equivalent.....10

Figure 5: The gethostbyaddr() function. On the left is a typical Berkeley Sockets source code fragment. On the right is the Windows Sockets equivalent.....12

Figure 6: Microsoft Developer Studio. Developer Studio provides a graphical user interface (GUI) to the compilers, online manuals, source code editor, and other tools that are included with Visual C++.....16

Figure 7: The layered design of a NetSolve Client. C and MATLAB Interface routines call Client Core Functions. Client Core Functions call Core Functions. This approach isolates most of the non-portable code in the core functions and makes adding new interfaces easier.....19

Figure 8: The MATLAB Command Window.....23

Figure 9: Example C Interface Call to NetSolve.....26

Figure 10: Example MATLAB Interface Call to NetSolve.....26

Figure 11: Creating the Netsolve.ini file. The Netsolve.ini file being created here defines the host "cetus5a.cs.utk.edu" as the NetSolve Agent.....28

Figure 12: The *Entry Point Not Found* Error Dialog. This dialog is automatically displayed when an implicitly linked dynamic link library can't be located at runtime.....28

Chapter 1

Introduction

1.1 Motivation

Software libraries have been developed to solve a variety of scientific computing problems. There are, however, barriers that limit the usefulness of these libraries. One such barrier is access. Libraries are seldom installed everywhere they are needed. Some libraries may only be available for certain platforms. Others may be highly optimized for specific platforms and, thus, may not perform well on local computing resources. When libraries have been installed, they may be out-of-date. Even if all the needed libraries are available and up-to-date, the performance of local computing resources may not be adequate to solve a very large problem (or set of problems) in a timely manner.

The NetSolve system [CD97] removes many of the traditional barriers that have limited the usefulness of software libraries. However, the NetSolve system itself is not available for all platforms. In particular, it is not available for the popular Windows NT and Windows 95 operating systems.

1.2 Objectives

The ultimate goal of this research is to make the NetSolve system available to Windows NT and Windows 95 users. This work has resulted in the creation of a NetSolve client for the Windows NT and Windows 95 operating systems. This client provides both a C language interface and an interface that can be used from MATLAB [Mat96]. To maximize compatibility, the client was developed using the industry standard Microsoft Visual C++ compiler [Vis96].

Since Dynamic Link Libraries (DLLs) are widely used in this environment, both the C and MATLAB interfaces are implemented as DLLs. The C interface is also available as a static link library. Finally, complete source code is available for both the C and MATLAB interfaces.

1.3 The NetSolve System

The NetSolve system allows users to access software libraries over a network. The NetSolve system is comprised of a loosely connected set of computers [CDS+98]. These computers may be located in the same building and connected by a small, private intranet. Conversely, the computers in a NetSolve system may be scattered about the globe and connected by the Internet.

Figure 1 shows a conceptual view of the NetSolve system. This figure depicts how problems are solved in two basic steps. In the first step, a NetSolve client sends a problem description to a NetSolve agent. The agent determines which NetSolve servers can solve the problem and returns this list of servers to the client. In the second step, the client sends the problem to the first server on the list returned in step one. This server solves the problem and returns the solution to the client. If the server is unavailable or unable to solve

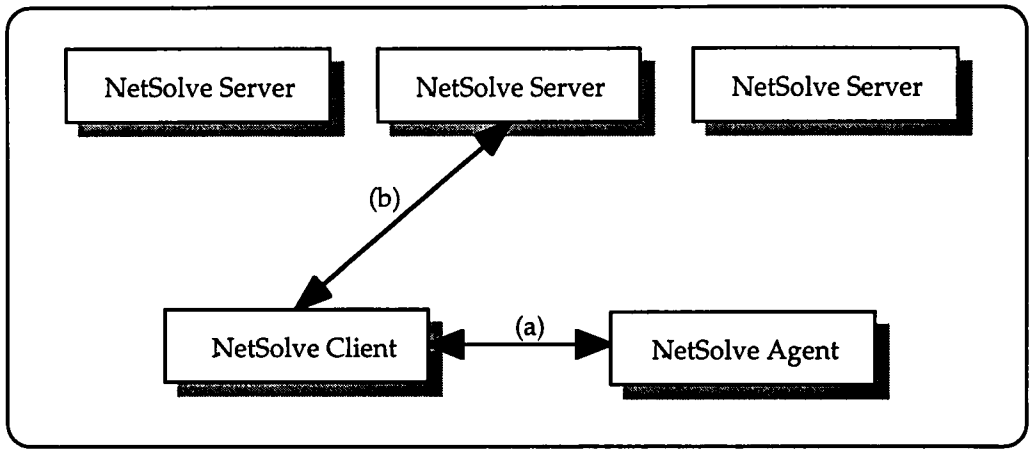


Figure 1: The NetSolve System. (a) A client sends a problem to an agent and receives a sorted list of servers that can solve the problem. (b) The client sends the problem to the first server in the list and receives the solution.

the problem for any reason, the client will send the problem to the next server in the list. This process will continue until the problem has been solved or until all the servers in the list have failed.

1.4 Related Work

1.4.1 Previous NetSolve Clients

NetSolve was originally implemented for the UNIX operating system [CD97]. This implementation now supports a variety of computing architectures (e.g., Sun Solaris, DEC Alpha, NeXT, SGI, IBM RS/6000, Cray T3E, and others)[CDS*98]. Interfaces have been developed for C, Fortran, MATLAB, Mathematica, and Java.

This implementation relies only on standard features of the UNIX operating system. Thus, it can easily be installed without the need to locate or purchase additional software components. However, this implementation is specific to the UNIX operating system. It cannot be used with other operating systems such as Windows NT, Windows 95, or the

Mac OS. The Java client does provide an interface for any user with a Java-enabled web browser. Although this is very useful, it provides only a limited level of access to non-UNIX hosts.

It is important to note that the work described in this document is based on the UNIX implementation described above. In fact, a goal in the development of the Windows NT/95 implementation was to reuse as much of the UNIX implementation as possible. Having a common core of shared code simplifies maintenance and helps to ensure that the Windows NT/95 implementation is kept up-to-date.

1.4.2 PVM Implementation for Windows

The Parallel Virtual Machine (PVM) is a system that combines a collection of heterogeneous computing resources into a single cooperative resource for parallel computing. Although not directly related to NetSolve, the PVM system does address many of the same design and implementation considerations. Like NetSolve, this system was originally implemented for the UNIX operating system and later implemented for the Windows 95 and Windows NT operating systems [FD96]. Furthermore, it relies on many of the same UNIX operating system features (e.g., a Sockets interface) that are required for NetSolve. Thus, the PVM implementation for Windows served as an excellent resource during the preparation of the work described in this document.

Chapter 2

Design Considerations

2.1 Overview

Since this work is based on a UNIX implementation (see section 1.4.1 above), most of the design considerations are related to differences between the UNIX and Windows environments. However, there are also design issues to consider that are related to the differences between Windows NT and Windows 95. Although they have much in common, Windows NT and Windows 95 are different operating systems. There is no guarantee that software written for Windows 95 will work with Windows NT (or vice versa). Care must be taken to design and develop software that will work with both Windows NT and Windows 95 [Vis96]. The remainder of this chapter will discuss some of the key issues that were considered in the design of a Windows NT/95 client for NetSolve.

2.2 Environment Variables, the Registry, and Initialization Files

Environment variables are used to define information about the local computing environment. Once defined, this information is generally available to any process that executes in that environment. A common example is the PATH environment variable. PATH typically contains a list of directories that should be searched when trying to locate a file. This search path would be available to any process that was executing in a context where PATH was defined.

Environment variables are widely used in UNIX operating systems. They are a reliable way to define information about the local computing environment. In the Windows operating systems, information about the local computing environment is often stored in a system-wide data repository called the *Registry*. However, it is difficult for the end user to access the Registry directly. The Registry is normally accessed from an application (or control panel) which provides a friendly user interface. It is also common to store information that is specific to an application in an initialization file. An initialization file is (most often) a text file that defines information that is needed during the initialization of an application.

Environment variables are supported in Windows operating systems (e.g., the PATH environment variable). However, the Windows operating systems have a very limited amount of memory available for storing environment variables. Furthermore, the amount of memory that will be available varies depending on the total amount of installed memory, the version of Windows being used, and user settings. Thus, when attempting to create an environment variable, it is not uncommon to encounter an "Out of Environment Space" error. This makes the use of environment variables in the Windows operating systems much less attractive than in the UNIX operating systems.

2.3 Sockets

Sockets, or Berkeley Sockets, is a networking API that is widely used in UNIX operating systems [Tan96]. It is often used with the TCP transport protocol, but is designed to support other protocols as well. Figure 2 (which is derived from Figure 6-6 in [Tan96]) lists the primitives that make up the basic Sockets API. These primitives allow an application to establish a network connection, send and receive data on that connection, and then release the connection.

The Windows operating systems also support the Sockets API. The Windows implementation is called Windows Sockets (often abbreviated to "WinSock") [HTA+93]. Although designed to be compatible with Berkeley Sockets, Windows Sockets deviates from the Berkeley standard in a number of instances. Some of these differences are merely cosmetic. For example, some primitives have the same function as their Berkeley counterpart, but have been given different names in Windows Sockets (e.g., Closesocket() instead of Close()). Other differences are more significant and will be discussed in the following sections (which were derived from [HTA+93]).

2.3.1 The SOCKET Data Type

A socket handle is returned by the Socket() and Accept() primitives and is used thereafter to identify the connection. In Berkeley Sockets, all socket handles are small non-negative integers and the integer constant "-1" is used to represent an invalid socket handle. A new data type, SOCKET, has been defined for socket handles in Windows Sockets and the symbolic constant INVALID_SOCKET is used to represent an invalid socket handle. It is not safe to assume that socket handles will be small non-negative integers nor to use the

Socket()	Create a new communication end point.
Bind()	Attach a local address to a socket.
Listen()	Announce willingness to accept connections.
Accept()	Block the caller until a connection attempt arrives.
Connect()	Actively attempt to establish a connection.
Send()	Send some data over the connection.
Recv()	Receive some data from the connection.
Close()	Release the connection.

Figure 2: Primitives in the Berkeley Sockets API.

integer constant "-1" to represent an invalid socket handle. Figure 3 shows a typical source code sequence in Berkeley Sockets and the equivalent source code sequence in Windows Sockets.

2.3.2 The Select() Function

The Select() function is used to determine the status of one or more sockets. For each socket, the caller may request information on read, write, or error status. The set of sockets for which a given status is requested is indicated by an "fd_set" structure.

Because socket handles are not represented by small non-negative integers, the implementation of the Select() function in the Windows Sockets API differs from the implementation in Berkeley Sockets. Each set of sockets is still represented by an "fd_set" structure, but is implemented as an array of SOCKETs instead of being stored as a bitmask. Thus, applications **must** adhere to the use of the "FD_XXX" macros to set, initialize, clear, and check "fd_set" structures.

Typical Berkeley Sockets Source Code Fragment:	**Typical Windows Sockets Source Code Fragment:**

int s;
...
s = socket(...);
if (s == -1) {
  /* error handling code goes here */
}
	SOCKET s;
	...
	s = socket(...);
	if (s == INVALID_SOCKET) {
	  /* error handling code goes here */
	}

Figure 3: The SOCKET Data Type and INVALID_SOCKET Constant. On the left is a typical Berkeley Sockets source code fragment. On the right is the Windows Sockets equivalent.

2.3.3 Error Codes

In Berkeley Sockets, API functions typically return "-1" to indicate an error and place an appropriate error code in a global variable called "errno". In the Windows Sockets implementation, API functions typically return the symbolic constant "SOCKET_ERROR" to indicate an error. Furthermore, error codes are **not** made available in the "errno" global variable. Instead, a new API function is introduced, WSAGetLastError(), to provide access to the error code from the last socket function. Figure 4 shows a typical source code sequence in Berkeley Sockets and the equivalent source code sequence in Windows Sockets.

Typical Berkeley Sockets Source Code Fragment:	**Typical Windows Sockets Source Code Fragment:**

r = recv(...);
if (r == -1 &&
errno == EWOULDBLOCK)
{...}
	r = recv(...);
	if (r == SOCKET_ERROR &&
	WSAGetLastError() == EWOULDBLOCK)
	{...}

Figure 4: The SOCKET_ERROR constant and WSAGetLastError() function. On the left is a typical Berkeley Sockets source code fragment. On the right is the Windows Sockets equivalent.

2.3.4 The WSAStartup() Function

The Berkeley Sockets API requires no explicit initialization prior to use. Presumably, the API is initialized during operating system startup. The Windows Sockets API, however, does require an initialization prior to use. The function WSAStartup() **must** be called before any other API function can be used. Calling another Windows Sockets API function before calling WSAStartup() will result in an immediate error return.

2.3.5 The gethostbyaddr() Function

The gethostbyaddr() function is used to get information about a host when the TCP/IP address is known. There are no notable differences between the Berkeley Sockets implementation and the Windows Sockets implementation for Windows NT. However, the Windows 95 implementation does behave differently. Specifically, the Windows 95 implementation behaves differently when the supplied TCP/IP address is zero.

Since it is more efficient to connect by address, it is preferable to connect to a host by name only if the TCP/IP address is not known. However, the TCP/IP address typically is not known unless a previous connection has been made to the same host. Thus, it is good practice to save the TCP/IP addresses of hosts that might be visited again in the same session. Then, before connecting to a host, gethostbyaddr() is called first with the saved TCP/IP address. If this fails, then gethostbyname() is called. If a host has not been visited before, its TCP/IP address is not known and is typically initialized to zero.

In both Berkeley Sockets and the Windows NT implementation of Windows Sockets, calling gethostbyaddr() with a TCP/IP address of zero results in an error. This error, as described in the sequence above, prompts a call to gethostbyname() and the connection process proceeds normally. However, the Windows 95 implementation of Windows Sockets **does not** return an error in this case. Instead, it returns a successful completion status but, of course, returns no host information. When using the connection sequence described above, this results in a connection failure.

Fortunately, the solution to this problem is very simple: do not call gethostbyaddr() if the TCP/IP address is zero. Figure 5 shows a typical source code sequence in Berkeley Sockets and the equivalent source code sequence in Windows Sockets. Again, the Berkeley Sockets version would work fine with the Windows NT implementation of Windows Sockets, but would not work with the Windows 95 implementation. The Windows Sockets version will work with either Berkeley Sockets or Windows Sockets.

Typical Berkeley Sockets Source Code Fragment:	**Typical Windows Sockets Source Code Fragment:**

hp = NULL;
...
hp = gethostbyaddr(...);
if (hp == NULL) {
  hp = gethostbyname(...);
  ...
}
	hp = NULL;
	...
	If (ipAddr != 0)
	  hp = gethostbyaddr(...);
	if (hp == NULL) {
	  hp = gethostbyname(...);
	  ...
	}

Figure 5: The gethostbyaddr() function. On the left is a typical Berkeley Sockets source code fragment. On the right is the Windows Sockets equivalent.

2.4 XDR

External Data Representation (XDR) is a standard way of encoding and decoding data so that it is portable between computing resources with different architectures [Tan96]. For example, all computers do not store the bytes that make up a multi-byte integer in the same order. There are two ways that a 16-bit integer might be stored. It might be stored with the low-order byte at the starting address (little endian) or with the high-order byte at the starting address (big endian) [Ste90]. More specifically, the Intel x86 and Pentium architectures use the "little endian" approach, while many other architectures (e.g., IBM 370, Motorola 68000, etc.) use "big endian".

A typical XDR implementation provides more than one destination option for encoding data (source option for decoding data). One common option is called "stdio" and allows data to be encoded to or decoded from files. Another common option is called "memory" and allows data to be encoded to or decoded from memory. Furthermore, the interface is extensible so that additional sources/destinations can be developed as needed.

Most UNIX operating system implementations provide an XDR API [Xdr90]. However, an XDR API is not typically provided with Windows NT or Windows 95.

2.5 Text Files

Clearly, text files are common to both UNIX operating systems and Windows operating systems. However, there is one notable difference in their implementation. The UNIX operating systems use a single character, the linefeed, to delimit the lines of text files. Conversely, the Windows operating systems use two characters for this purpose, the carriage return/linefeed pair. For completeness, it should also be noted that the Macintosh operating system uses a single character for this purpose, like UNIX. Unfortunately this character is the carriage return instead of the linefeed.

2.6 Development Environment and Tools

UNIX operating systems include a number of software development tools. These tools include compilers and linkers (e.g., "cc"), a utility for evaluating dependencies and conditionally invoking compilers and linkers (e.g., "make"), online reference manuals (e.g., "man"), and so on [Ore90]. These tools are not typically included with the Windows NT or Windows 95 operating system implementations, but are available separately. Furthermore, although similar tools are available, these tools are not identical to their UNIX counterparts.

Chapter 3

Implementation Issues

3.1 Overview

Implementation was guided by several high-level goals. The first goal, as discussed earlier (see section 1.4.1), was to reuse as much of the original UNIX implementation as possible. A second goal was to use only widely available, industry standard tools. This helps to ensure that Windows NT and Windows 95 users will have access to the tools used in this work. A third goal was to provide deliverables is several formats to fill the needs of different types of users. Specifically, the complete source code should be made available for users who might wish to customize or integrate NetSolve into another application. Static link libraries should be made available to users who might wish to build stand-alone applications that use NetSolve. Finally, dynamic link libraries (DLLs) should be provided for users who might prefer the advantages of DLLs (see section 3.4.3).

The design considerations discussed in chapter 2 represent important issues that must be dealt with during implementation. The remainder of this chapter will address these issues as well as the goals described above.

3.2 Development Environment and Tools

Microsoft Visual C++ is the de facto standard for developing C and C++ applications for the Windows operating systems and, thus, was selected for use in developing NetSolve for Windows. Visual C++ is not bundled with the Windows operating systems, but must be purchased and installed separately. Figure 6 shows the Microsoft Developer Studio, which is bundled with Visual C++. Developer Studio provides a graphical user interface (GUI) to the compilers, online manuals, source code editor, and many of the other tools that are included with Visual C++. Developer Studio was used extensively to search the online documentation, but was not used to access the compiler and other tools. As is evident from Figure 6, most of the interface is consumed by menus, buttons, icons, scroll bars, tabs, and so on. The user is left with only a small fraction of the available space in which to actually work (e.g., view online manuals or edit source code). Thus, a simple text editor was used to edit source code and the command line versions of the Visual C++ tools were used whenever possible.

UNIX-like "make" files were developed to automate the building of each component of NetSolve for Windows. The Visual C++ utility that corresponds to the UNIX "make" utility is called "nmake." Furthermore, a Windows batch file was developed (make.bat) that automates the build process by invoking the nmake utility with each make file to build all the components of NetSolve for Windows. Thus, the user can build all the components of NetSolve for Windows by executing the "make.bat" batch file, or can build an individual component by executing the "nmake" utility with the make file for that component.

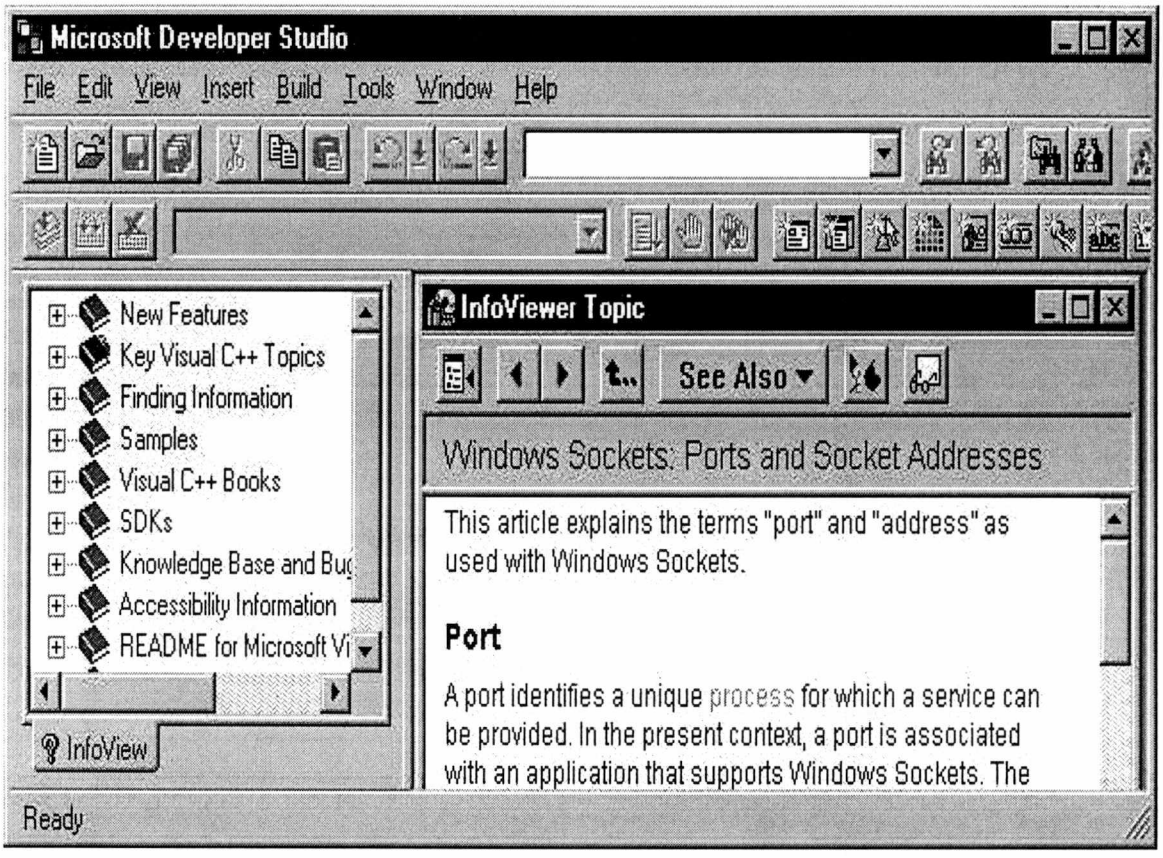


Figure 6: Microsoft Developer Studio. Developer Studio provides a graphical user interface (GUI) to the compilers, online manuals, source code editor, and other tools that are included with Visual C++.

3.3 XDR

As was discussed in section 2.4, the Windows operating systems do not provide an XDR API. Furthermore, an XDR API is not provided with Microsoft Visual C++. A search of the online manuals in Microsoft Developer Studio (see section 3.2 above) for "XDR" found no matches. Some research into the PVM implementation for Windows (see section 1.4.2) revealed that a custom XDR implementation was used. Unfortunately, only a small subset of XDR (the subset needed by PVM) had been implemented. This subset did not include all the parts of XDR that were being used by NetSolve. By searching the Internet, a complete XDR implementation for Windows NT was eventually located. This implementation, developed by Martin F. Gergeleit in 1993, was adapted from an original implementation by Sun Microsystems.

This implementation of XDR proved to work quite well. Because the implementation was a bit dated, some effort was required (e.g., updating function prototypes, type casting, etc.) to make it acceptable to more modern C compilers. The goal in updating these routines was to make only those changes that were required to compile cleanly with modern C compilers.

Early versions of NetSolve took advantage of an XDR feature that is specific to UNIX operating systems. In UNIX systems, socket handles can generally be used anywhere file handles can be used. That is, the same functions often work with either sockets or files (e.g., read(), write(), and close()). Early versions of NetSolve took advantage of this commonality by using the XDR routines for "stdio" (which are intended for use with files) to do XDR conversions directly to and from sockets. Unfortunately, socket handles and file handles are **not** interchangeable in the Windows operating systems. Thus, the XDR routines for "stdio" could not be used with sockets.

To solve this problem, a new XDR interface was developed specifically for sockets. This interface was based on the existing implementation for "stdio" and was integrated into the complete XDR implementation for Windows described above. This new XDR interface was used successfully in the initial implementation of NetSolve for Windows. Later versions of NetSolve have been updated to use the XDR routines for "memory" instead of those for "stdio". This solution should be much more portable. Subsequently, NetSolve for Windows has also been updated to use the XDR routines for "memory" instead of the custom "socket" routines described above. The "socket" XDR routines are still included, however, since they may prove useful to others who might need to port UNIX code to the Windows environment.

3.4 C Interface

Once a development environment had been selected (see section 3.2) and an XDR implementation was in place (see section 3.3), a C interface was developed for NetSolve for Windows. The approach used was to start with the source code for the UNIX C interface and update it as needed for use with Windows. Fortunately, the UNIX source code for NetSolve employs a layered design. This design encapsulates less portable code in the lowest layers. Higher layers use the APIs of the lower layers and are oblivious to their implementation details. Figure 7 (which is derived from figure 1.2 in [CDS+98]) illustrates this layering for NetSolve clients.

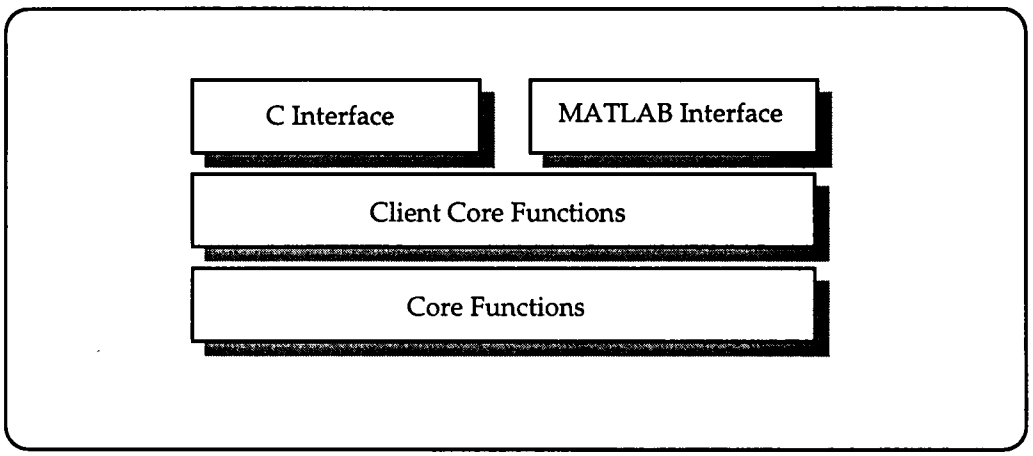


Figure 7: The layered design of a NetSolve Client. C and MATLAB Interface routines call Client Core Functions. Client Core Functions call Core Functions. This approach isolates most of the non-portable code in the core functions and makes adding new interfaces easier.

3.4.1 Implementing an Initialization File

The UNIX NetSolve client used an environment variable to define the TCP/IP name of the NetSolve agent. As was discussed in section 2.2, use of environment variables is problematic with Windows operating systems. Use of the Registry was considered, but was rejected because of the poor user interface. Instead, a simple initialization file was used to define the TCP/IP address of the agent. The getNetsolveAgent() routine (a core function) was rewritten to read the TCP/IP name from the initialization file rather than obtain it from an environment variable.

3.4.2 Implementing Windows Sockets

Differences between Berkeley Sockets and Windows Sockets, as well as differences between Windows NT and Window 95 implementations of Windows Sockets were discussed in section 2.3. The techniques outlined in section 2.3 were used to update the UNIX NetSolve source code to work with Windows Sockets. Because of the layered

design of the UNIX NetSolve source code, almost all of the Sockets code is contained in only three source files: communicator.c, clientutil.c, and socketutil.c. See appendices C, D, and E, respectively, for a complete listing of each of these files.

In addition, calls to write() in the UNIX source were replaced with calls to a new routine called SendData(). Likewise, calls to tread() were replaced with calls to a new routine called ReceiveData(). These new routines offer some additional capabilities. For example, both SendData() and ReceiveData() will time out and return an error after a period of inactivity on the socket. The time out period, defined by the symbolic constant SOCKET_TIMEOUT_SECONDS, is set to two minutes.

Finally, a new routine called WinSockInit() was written which initializes the Windows Sockets API by calling WSAStartup(). Calls to WinSockInit() were added to the external client interface routines (e.g., netsl()) to ensure that the Windows Sockets interface was initialized prior to use. The new WinSockInit() routine, as well as the new SendData() and ReceiveData() routines described above, is located in a new source file called "win_util.c". See Appendix B for a complete listing of win_util.c.

3.4.3 Implementing Static and Dynamic Link Libraries

The C interface was first implemented by linking the individual object files that make up the NetSolve client with a test program that included a main() function to build a standalone executable. This approach was convenient for debugging because individual NetSolve source files could be recompiled and linked to build an executable without having to rebuild any libraries. Once the C interface was working reliably, a static link library (libnetsolve.lib) was built using the "lib" utility that is bundled with Visual C++. The C interface test program was then linked with this library to build a standalone executable.

Finally, a dynamic link library (netsolve.dll) was built using the "link" utility that is bundled with Visual C++.

To support the dynamic link library, an additional source file (netsolvedll.c) was created. This source file contains a new function called DllEntryPoint() that serves as the entry point for the dynamic link library. Another new source file (netsolvedll.def) is used to define the functions that the dynamic link library will export. This file is used by the link utility when creating the dynamic link library. See Appendix A for a complete listing of the netsolvedll.c and netsolvedll.def source files.

There are two approaches to using a dynamic link library: implicit linkage and explicit linkage [Kru96]. Implicit linkage is very easy to use and, thus, is the preferred method for most situations. When a dynamic link library is created, the linker produces a companion **import** library. This import library contains the name of the DLL and its exported symbols, but does not contain any of the code. If this import library is included when building an executable, the dynamic link library will be implicitly linked to the executable. When the executable starts up, it will load and link to the dynamic link library.

Explicit linkage is more difficult to implement, but allows the user more control. With implicit linkage, the dynamic link library is automatically loaded and every exported function is bound during startup of the executable. With explicit linkage, the user decides when (and if) the dynamic link library will be loaded and which exported functions to bind. The LoadLibrary() function is used to load the dynamic link library and the GetProcAddress() function is used to bind an exported function. Explicit linkage might be preferred in cases where NetSolve is used only by one part of an application. In this case, it might be preferable to wait until a user enters the part of the application that needs NetSolve before loading the dynamic link library. Explicit linkage might also be preferred

in situations where resources are tight and dynamic link libraries must be carefully loaded and unloaded to avoid exceeding the available memory.

3.5 MATLAB Interface

Like Visual C++, MATLAB is not bundled with the Windows operating systems. It must be purchased and installed separately. When executed, MATLAB for Windows opens a command window similar to the one shown in Figure 8. Thus, the user interface to MATLAB is command driven, just as it is on UNIX systems.

NetSolve's MATLAB interface reuses much of the code that was implemented for the C interface. As with the C interface, calls to WinSockInit() were added to the external client interface routines (e.g., mxnetsolve.c) to ensure that the Windows Sockets interface was initialized prior to use. Because of the layered design (described in section 3.4 above) used for the NetSolve client, no further modifications to the UNIX source code were required to implement the MATLAB interface.

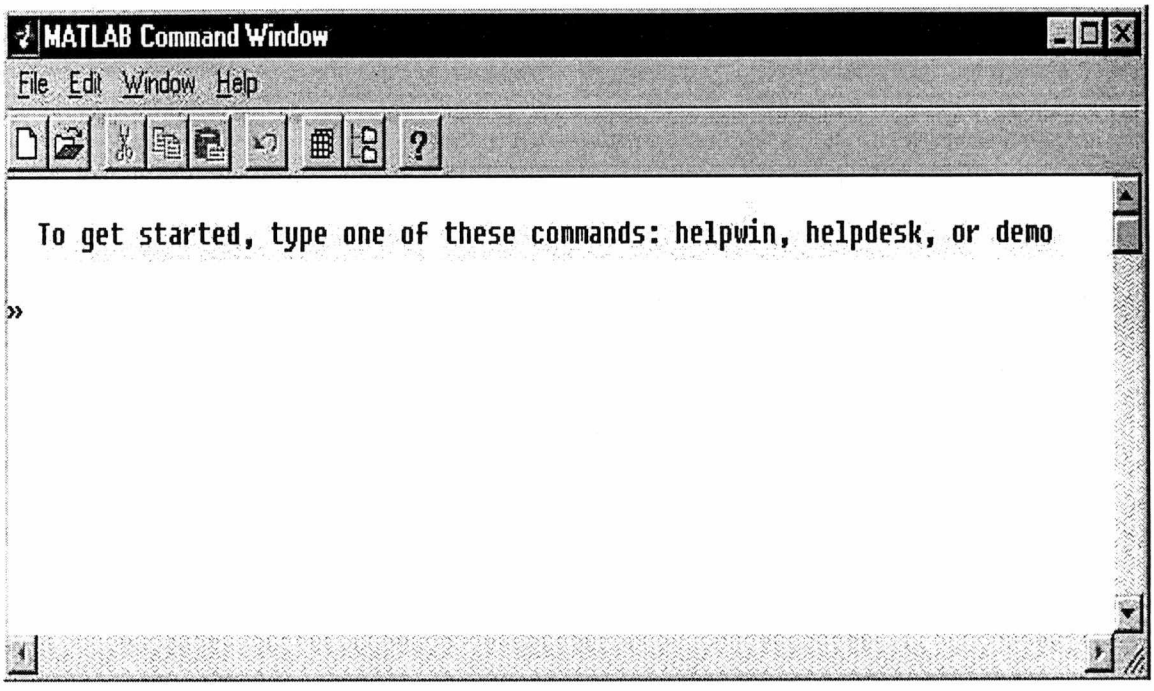


Figure 8: The MATLAB Command Window.

Chapter 4

Using NetSolve For Windows

4.1 Overview

This chapter discusses issues related to using the NetSolve client for Windows NT and Windows 95. First, issues related to downloading and installing the software are discussed. Then, issues related to using the software are discussed and some examples are provided.

4.2 Downloading and Installing the Software

The NetSolve for Windows software has been integrated with the NetSolve for UNIX distribution. The latest version of this distribution is available for download from the NetSolve web page at: http://www.cs.utk.edu/netsolve. Because NetSolve is an ongoing research project, this distribution is likely to be in a state of flux. Thus, it is not feasible to include detailed information in this thesis about the current download and installation procedure. Instead, the user should follow the instructions on the web page, in the most current *NetSolve User's Guide* (which is available for download from the web page), and in any "readme" files that are distributed with the software.

4.3 Using the Software

4.3.1 General Information

Once again, it is not feasible to include detailed information about using NetSolve in this thesis because this information is subject to change. Instead, brief examples will be presented that illustrate how NetSolve is called from both the C interface and the MATLAB interface. The remaining sections of this chapter will focus on NetSolve features that are specific to the Windows implementation. General information about using NetSolve is documented in the *NetSolve User's Guide*. The most current version of this document is available for download from the NetSolve web page (see section 4.2 above).

Figure 9 shows an example C interface call to NetSolve. In this example, NetSolve is used to sort an array of doubles. The first argument to the "netsl()" function identifies the problem to be solved as "dqsort", a quicksort function that works with doubles. The other arguments are inputs to and outputs from the problem. The "netsl()" call sends a blocking request to NetSolve. That is, the calling process will block until the problem is solved and the results have been returned. A non-blocking interface is also available by calling "netslnb()" instead of "netsl()."

Figure 10 shows an example MATLAB interface call to NetSolve. In this example, NetSolve is used to invoke the "linsol" function. Note that there is no need to pass the sizes of the arguments "a" and "b" since this information is part of the variable's definition in MATLAB. In the C interface example (see figure 9), it was necessary to pass the size of the array to be sorted.

```
double unsorted[4];
double sorted[4];
int status;
...

status = netsl("dqsort()",4,unsorted,sorted);
if (status < 0) {
   /* error handling code goes here */
}
```

Figure 9: Example C Interface Call to NetSolve.

```
a = rand(4);
b = rand(4,1);

[x] = netsolve('linsol',a,b);
[e] = netsolve_err;
if (e < 0)
   /* error handling code goes here */
end
```

Figure 10: Example MATLAB Interface Call to NetSolve.

4.3.2 Creating Netsolve.ini

NetSolve for Windows looks for a file called "Netsolve.ini" in the same directory as the executable. The "Netsolve.ini" file is a text file and should contain only one line. This line is an entry that identifies the agent the NetSolve client will use. Figure 11 shows an example "Netsolve.ini" file being created with the "Notepad" accessory that is bundled with Windows. In this example, the NetSolve agent is being defined as the host "cetus5a.cs.utk.edu."

4.3.3 Using Dynamic Link Libraries

When dynamic link libraries are used, the resulting application is divided into two (or more) parts: the executable (which usually is given an extension of ".exe") and the dynamic link library or libraries (which are usually given extensions of ".dll"). Thus, it is possible that the executable may be unable to locate the dynamic link library at runtime. If an implicitly linked (see section 3.4.3) dynamic link library cannot be found at runtime, the user will get an error dialog similar to the one shown in figure 12. Before returning this error, Windows will try to locate the dynamic link library by looking in: (1) the directory containing the executable, (2) the process's current directory, (3) the Windows system directory, (4) the Windows directory, and finally (5) the directories listed in the PATH environment variable [Kru96]. If the dynamic link library is explicitly linked, the caller has the option of specifying a full pathname in the call to LoadLibrary(). Unless a full pathname is specified, the sequence of directories listed above is searched for the dynamic link library. If the library is not found, the dialog in figure 12 **will not** automatically be displayed. Instead, the LoadLibrary() call will return with an error status and the caller must handle the error.

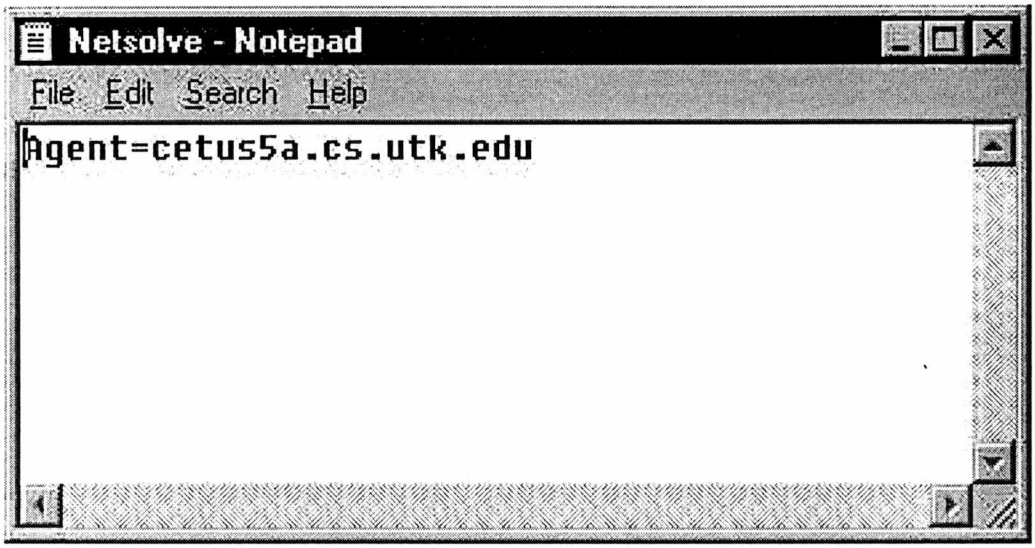


Figure 11: Creating the Netsolve.ini file. The Netsolve.ini file being created here defines the host "cetus5a.cs.utk.edu" as the NetSolve Agent.

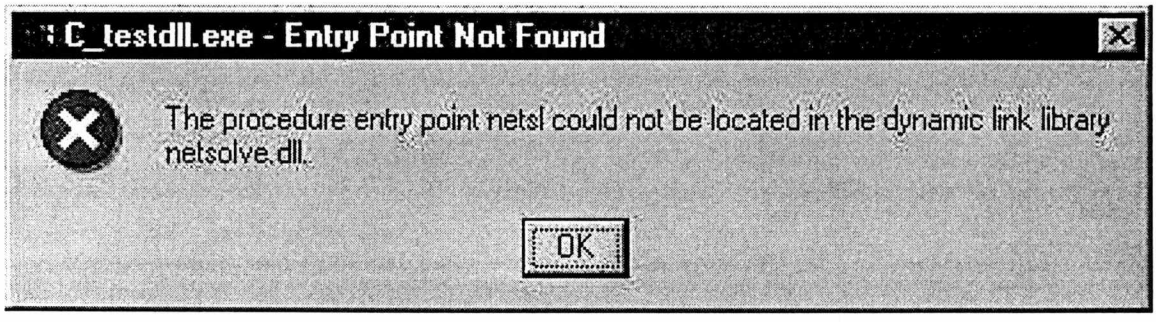


Figure 12: The *Entry Point Not Found* Error Dialog. This dialog is automatically displayed when an implicitly linked dynamic link library cannot be located at runtime.

4.3.5 Using UPF Files

NetSolve includes a facility whereby a user provided function (UPF) can be sent, as part of a problem, to a NetSolve server [CDS+98]. Some computational library routines require the user to provide a function (or multiple functions) in addition to numerical data (e.g., matrices, vectors, etc.). These UPFs are stored as text files on the local user's system and are transmitted to the NetSolve server along with the numerical data. The problem, which is discussed in section 2.5, is that the lines in text files are delimited differently on UNIX systems than on Windows systems. The solution is to convert UPF files (or a copy of those files) to the UNIX text file format. Utilities are readily available that convert the line endings in text files to and/or from the various operating system standards.

Chapter 5

Conclusions and Future Work

The primary task undertaken in this thesis was the development of a NetSolve client for Windows NT and Windows 95. The existing NetSolve client for UNIX was used as a starting point and was modified to work in the Windows environment. An XDR implementation was located, adapted for use in the Windows environment, and extended to provide direct support for sockets. Industry standard software development tools were used in the development of this work and deliverables were provided in the most popular formats (e.g., dynamic link libraries).

Some enhancements could be made in the course of future work to extend the usefulness of the NetSolve for Windows client. The primary avenue for enhancement would be to add additional interfaces. For consistency with the UNIX client, interfaces might be added for Fortran and Mathemetica. In addition, a Visual BASIC interface might be worthwhile due to the popularity of that language in the Windows environment.

Bibliography

Bibliography

[CD97] Henri Casanova and Jack Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. Published in the International Journal of Supercomputer Applications and High Performance Computing, 11(3):212-223, 1997.

[Mat96] MATLAB Application Program Interface Guide. The MATH WORKS, Inc., 1996.

[Vis96] Visual C++ User's Guide. Microsoft Corporation, 1996.

[CDS+98] Henri Casanova, Jack Dongarra, Keith Seymour, and Alexander Karainov. User's Guide to NetSolve. September, 1998.

[FD96] Markus Fischer and Jack Dongarra. Another Architecture: PVM on Windows 95/NT. October, 1996.

[Tan96] Andrew S. Tanenbaum. *Computer Networks, Third Edition.* Prentice Hall, 1996.

[HTA+93] Windows Sockets: An Open Interface for Network Programming under Microsoft Windows. January, 1993.

[Ste90] W. Richard Stevens. *UNIX Network Programming.* Prentice Hall, 1990.

[Xdr90] External Data Representation: Sun Technical Notes. Sun Microsystems, 1990.

[Ore90] Tim O'Reilly. UNIX in a Nutshell. 1990.

[Kru96] David J. Kruglinski. *Inside Visual C++.* Microsoft Press, 1996.

Appendices

Appendix A: Source Code for DLL Routines

```
/*************************************************************************/
/*  netsolvedll.c                                                        */
/*  Implement a Win32 DLL for Netsolve.                                  */
/*                                                                       */
/*  by Rick Phillips                                                     */
/*************************************************************************/

#include <windows.h>
#include "netsolve.h"

/*************************************************************************
    DllEntryPoint
    The entry point of the DLL.
*************************************************************************/
BOOL WINAPI DllEntryPoint(HINSTANCE hDLL, DWORD dwReason, LPVOID Reserved)
{
    switch (dwReason) {
        case DLL_PROCESS_ATTACH:
            break;
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}



;*************************************************************************/
;*  netsolvedll.def                                                      */
;*  Implement a Win32 DLL for Netsolve.                                  */
;*                                                                       */
;*  by Rick Phillips                                                     */
;*************************************************************************/

LIBRARY netsolve

CODE    PRELOAD MOVEABLE DISCARDABLE
DATA    PRELOAD SINGLE

EXPORTS
    ; The names of the DLL functions go here
    netsl
    netslnb
    netslwt
    netslpr
    netslerr
    netsl_farm
```

Appendix B: Source Code for win_util.c

```
/*************************************************************/
/*  win_util.c                                               */
/*  by Rick Phillips                                         */
/*************************************************************/

/* Include files */
#include <stdio.h>
#include "general.H"
#include "win_util.h"

/* Global variables */
WSADATA WSAData;
static int init = 0;



/*****************************************************************************
    WinSockInit

    Purpose:
        Initialize the WinSock routines.  Call this routine before making any other
        WinSock calls.

    Argument List:
        void

    Returns:
        TRUE on success, FALSE on failure.
*****************************************************************************/
int WinSockInit(void)
{
    /* Return success if WinSock has already been initialized */
    if (init++)
        return(TRUE);

    /* Initialize the WinSock library */
    if (WSAStartup(0x0101, &WSAData)) {
        init = 0;
        fprintf(STDERR"WinSockInit: WSAStartup failed.\n");
        WSACleanup();
        return(FALSE);
    }

    /* Success! */
    return(TRUE);
}
```

```
/*****************************************************************************
    WinSockFinish

    Purpose:
        Release the WinSock library.  Call this routine when you are finished with all
        WinSock calls.

    Argument List:
        void

    Returns:
        TRUE on success, FALSE on failure.
*****************************************************************************/
int WinSockFinish(void)
{
    /* Return success if WinSock had doesn't need to be released */
    if (init == 0 || --init > 0)
        return(TRUE);

    /* Release WinSock */
    if (WSACleanup()) {
        init = 0;
        fprintf(STDERR"WinSockFinish: WSACleanup failed.\n");
        return(FALSE);
    }

    /* Success! */
    return(TRUE);
}


/*****************************************************************************
    SendData

    Purpose:
        Send data on an open socket.  This is a synchronous routine with a timeout.
        Control isn't returned until all the data has been sent or the timeout value is
        exceeded.

    Argument List:
        I   theSocket       socket to use.
        I   buffer          pointer to the buffer containing the data to be sent.
        I   bufferSize      number of bytes to send.

    Returns:
        Success         Actual number of bytes sent (will be equal to bufferSize).
        Timeout         Actual number of bytes sent (will be less than bufferSize,
                        but >= 0).
        Socket Error    SOCKET_ERROR.  Use WSAGetLastError() to get the actual error.
*****************************************************************************/
long SendData(SOCKET theSocket, void *buffer, long bufferSize)
{
    long            status = 1;
    long            bytesSent = 0;
    long            byteCount;
    fd_set          fdSet, fdMasterSet;
    struct timeval  timeOut;
```

```
    /* Initialize the timeout value */
    timeOut.tv_sec = SOCKET_TIMEOUT_SECONDS;
    timeOut.tv_usec = 0;

    /* Initialize the master fd set */
    FD_ZERO(&fdMasterSet);
    FD_SET(theSocket, &fdMasterSet);

    /* Loop sending chunks of data until all has been sent */
    while (bytesSent < bufferSize) {

        /* Reset the fd set */
        fdSet = fdMasterSet;

        /* Wait until the socket is ready or until we time out */
        status = select(theSocket+1, NULL, &fdSet, NULL, &timeOut);

        switch (status) {
            case 1:     /* Socket is ready */
                /* Write a chunk of data */
                if ((byteCount = send(theSocket, ((char far *)buffer)+bytesSent,
                    bufferSize-bytesSent, 0)) == SOCKET_ERROR) {
                    fprintf(STDERR"SendData failed (socket error on send)\n");
                    return(SOCKET_ERROR);
                }
                /* Update the counter */
                bytesSent += byteCount;
                break;

            case 0:     /* Socket timed out */
                fprintf(STDERR"SendData failed (time out)\n");
                return(bytesSent);
                break;

            case SOCKET_ERROR:
                fprintf(STDERR"SendData failed (socket error on select)\n");
                return(SOCKET_ERROR);
                break;

            default:    /* Unknown error */
                fprintf(STDERR"SendData failed (unknown error)\n");
                return(SOCKET_ERROR);
                break;
        }
    }

    return(bytesSent);
}
```

```
/*****************************************************************************
    ReceiveData

    Purpose:
        Receive data on an open socket.  This is a synchronous routine with a timeout.
        Control isn't returned until all the data has been received or the timeout value is
        exceeded.

    Argument List:
        I   theSocket       socket to use.
        I   buffer          pointer to the buffer to copy the received data into.
        I   bufferSize      number of bytes to receive.

    Returns:
        Success         Actual number of bytes received (will be equal to bufferSize).
        Timeout         Actual number of bytes received (will be less than bufferSize,
                        but >= 0).
        Socket Error    SOCKET_ERROR.  Use WSAGetLastError() to get the actual error.
*****************************************************************************/
long ReceiveData(SOCKET theSocket, void *buffer, long bufferSize)
{
    int             status = 1;
    long            bytesReceived = 0;
    long            byteCount;
    fd_set          fdSet, fdMasterSet;
    struct timeval  timeOut;

    /* Initialize the timeout value */
    timeOut.tv_sec = SOCKET_TIMEOUT_SECONDS;
    timeOut.tv_usec = 0;

    /* Initialize the master fd set */
    FD_ZERO(&fdMasterSet);
    FD_SET(theSocket, &fdMasterSet);

    /* Loop receiving chunks of data until all has been received or waitSecs is exceeded */
    while ((bytesReceived < bufferSize) && (status > 0)) {

        /* Reset the fd set */
        fdSet = fdMasterSet;

        /* Wait until the socket is ready or until we time out */
        status = select(theSocket+1, &fdSet, NULL, NULL, &timeOut);

        switch (status) {
            case 1:     /* Socket is ready */
                /* Read a chunk of data */
                if ((byteCount = recv(theSocket, (char far *)buffer+bytesReceived,
                    bufferSize-bytesReceived, 0)) == SOCKET_ERROR) {
                    fprintf(STDERR"ReceiveData failed (socket error on recv)\n");
                    return(SOCKET_ERROR);
                }
                /* Update the counter */
                bytesReceived += byteCount;
                break;
```

```
            case 0:     /* Socket timed out */
                fprintf(STDERR"ReceiveData failed (time out)\n");
                return(bytesReceived);
                break;

            case SOCKET_ERROR:
                fprintf(STDERR"ReceiveData failed (socket error on select)\n");
                return(SOCKET_ERROR);
                break;

            default:    /* Unknown error */
                fprintf(STDERR"ReceiveData failed (unknown error)\n");
                return(SOCKET_ERROR);
                break;
        }
    }

    return(bytesReceived);
}


/*****************************************************************************
    bcopy

    Purpose:
        Copy memory.

    Argument List:
        I   s       pointer to source.
        O   d       pointer to destination.
        I   len     number of bytes to copy.

    Returns:
        void
*****************************************************************************/
void bcopy(const void *s, void *d, int len)
{
    for(; len > 0; len--)
        *((char *)d)++ = *((char *)s)++;
}
```

```
/*****************************************************************************
    bzero

    Purpose:
        Initialize memory to zero.

    Argument List:
        O   t       pointer to target.
        I   len     number of bytes to initialize.

    Returns:
        void
*****************************************************************************/
void bzero(void *t, int len)
{
    for(; len > 0; len--)
        *((char *)t)++ = (char) 0;
}



/*****************************************************************************
    bcmp

    Purpose:
        Compare memory.

    Argument List:
        I   m1      pointer to first memory block.
        I   m2      pointer to second memory block.
        I   len     number of bytes to compare.

    Returns:
        void
*****************************************************************************/
int bcmp(const void *m1, const void *m2, int len)
{
    for(; len > 0; len--, ((char *)m1)++, ((char *)m2)++)
        if (*((char *)m1) != *((char *)m2))
            return 1;
    return 0;
}
```

Appendix C: Source Code for communicator.c

```
/*************************************************************************/
/*  communicator.c                                                       */
/*                                                                       */
/*  Based on original work by Henri Casanova                             */
/*  Win32 version by Rick Phillips                                       */
/*************************************************************************/

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <io.h>
#include "types.h"
#include "xdr.H"
#include "core.h"

bool_t xdr_scomplex(XDR *,scomplex *);
bool_t xdr_dcomplex(XDR *,dcomplex *);

/* Global variables for the xdr sizes */
int XDR_SIZEOF_INT;
int XDR_SIZEOF_FLOAT;
int XDR_SIZEOF_DOUBLE;
int XDR_SIZEOF_CHAR;
int XDR_SIZEOF_SCOMPLEX;
int XDR_SIZEOF_DCOMPLEX;
int XDR_SIZEOF_U_CHAR;



/*************************************************************************
    setXDRSizes
    Computes the sizes of diverse data types once they are
    XDR encoded. Should be done with the function xdr_sizeof(),
    but it is not available on all platforms (e.g. ALPHA)
*************************************************************************/
int setXDRSizes()
{
    XDR xdr_stream;
    char buffer[128];
    int cc=1;
    int t_int=2;
    float t_float=2.0;
    double t_double=2.0;
    scomplex t_scomplex;
    dcomplex t_dcomplex;
    char t_char='a';
    unsigned char t_u_char='a';

    t_scomplex.r = 2.0;
    t_scomplex.i = 2.0;
    t_dcomplex.r = 2.0;
    t_dcomplex.i = 2.0;

    xdrmem_create(&xdr_stream,buffer,128,XDR_FREE);
```

```
    xdrmem_create(&xdr_stream,buffer,128,XDR_ENCODE);
    cc = cc && xdr_int(&xdr_stream,&t_int);
    XDR_SIZEOF_INT = xdr_getpos(&xdr_stream);
    xdr_destroy(&xdr_stream);

    xdrmem_create(&xdr_stream,buffer,128,XDR_ENCODE);
    cc = cc && xdr_float(&xdr_stream,&t_float);
    XDR_SIZEOF_FLOAT = xdr_getpos(&xdr_stream);
    xdr_destroy(&xdr_stream);

    xdrmem_create(&xdr_stream,buffer,128,XDR_FREE);
    xdrmem_create(&xdr_stream,buffer,128,XDR_ENCODE);
    cc = cc && xdr_double(&xdr_stream,&t_double);
    XDR_SIZEOF_DOUBLE = xdr_getpos(&xdr_stream);
    xdr_destroy(&xdr_stream);

    xdrmem_create(&xdr_stream,buffer,128,XDR_FREE);
    xdrmem_create(&xdr_stream,buffer,128,XDR_ENCODE);
    cc = cc && xdr_char(&xdr_stream,&t_char);
    XDR_SIZEOF_CHAR = xdr_getpos(&xdr_stream);
    xdr_destroy(&xdr_stream);

    xdrmem_create(&xdr_stream,buffer,128,XDR_FREE);
    xdrmem_create(&xdr_stream,buffer,128,XDR_ENCODE);
    cc = cc && xdr_scomplex(&xdr_stream,&t_scomplex);
    XDR_SIZEOF_SCOMPLEX = xdr_getpos(&xdr_stream);
    xdr_destroy(&xdr_stream);

    xdrmem_create(&xdr_stream,buffer,128,XDR_FREE);
    xdrmem_create(&xdr_stream,buffer,128,XDR_ENCODE);
    cc = cc && xdr_dcomplex(&xdr_stream,&t_dcomplex);
    XDR_SIZEOF_DCOMPLEX = xdr_getpos(&xdr_stream);
    xdr_destroy(&xdr_stream);

    xdrmem_create(&xdr_stream,buffer,128,XDR_FREE);
    xdrmem_create(&xdr_stream,buffer,128,XDR_ENCODE);
#ifndef PLAT_T3E
    cc = cc && xdr_u_char(&xdr_stream,&t_u_char);
#else
    cc = cc && xdr_u_char(&xdr_stream,(char *)(&t_u_char));
#endif
    XDR_SIZEOF_U_CHAR = xdr_getpos(&xdr_stream);
    xdr_destroy(&xdr_stream);

    if (cc == 0) {
        ns_errno = NetSolveSystemError;
        return -1;
    }
    else
        return 1;
}
```

```
/*****************************************************************************
    xdr_scomplex
    XDR encode a single precision complex
*****************************************************************************/
bool_t xdr_scomplex(XDR *xdrptr,scomplex *ptr)
{
    return (xdr_float(xdrptr,&ptr->r) && xdr_float(xdrptr,&ptr->i));
}



/*****************************************************************************
    xdr_dcomplex
    XDR encode a double precision complex
*****************************************************************************/
bool_t xdr_dcomplex(XDR *xdrptr,dcomplex *ptr)
{
    return (xdr_double(xdrptr,&ptr->r) && xdr_double(xdrptr,&ptr->i));
}



/*****************************************************************************
    newCommunicator
    Allocate the space for the communicator and assigns the fields
*****************************************************************************/
Communicator *newCommunicator(SOCKET sock,int encoding)
{
    Communicator *comm;
    static int firsttime=1;

    if (encoding == DATA_XDR) {
        if (firsttime) {
            setXDRSizes();
            firsttime = 0;
        }
    }
    comm = (Communicator *)calloc(1,sizeof(Communicator));
    comm->sock = sock;
    comm->encoding = encoding;
    return comm;
}
```

```
/**************************************************************************
    initTransaction
    Initiates a communication by sending a u_short under the network format
    to signal if XDR is being used or not. Returns a new communicator.
**************************************************************************/
Communicator *initTransaction(SOCKET sock,int encoding)
{
    u_long net_long;

    net_long = htonl((u_long)encoding);

    if (SendData(sock,&net_long,sizeof(u_long)) != sizeof(u_long)) {
        ns_errno = NetSolveNetworkError;
        return NULL;
    }
    else
        return newCommunicator(sock,encoding);
}



/**************************************************************************
    acceptTransaction
    Receives a u_short and determines the encoding.
    Returns a new communicator.
**************************************************************************/
Communicator *acceptTransaction(SOCKET sock)
{
    u_long net_long;
    int cc;
    int encoding;

    cc = ReceiveData(sock,(char *)(&net_long),sizeof(u_long));
    if (cc != sizeof(u_long)) {
        ns_errno = NetSolveNetworkError;
        return NULL;
    }
    encoding = (int)ntohl(net_long);
    return newCommunicator(sock,encoding);
}



/**************************************************************************
    endTransaction
    Terminates a transaction, kill the connection end free the Communicator.
**************************************************************************/
void endTransaction(Communicator *comm)
{
    closesocket(comm->sock);
    free(comm);
}
```

```
/*************************************************************************
    netsolve_sizeof
    Returns the size in bytes of Netsolve data types.
*************************************************************************/
int netsolve_sizeof(int data_type)
{
    switch(data_type) {
        case NETSOLVE_I:
            return sizeof(int);
        case NETSOLVE_D:
            return sizeof(double);
        case NETSOLVE_S:
            return sizeof(float);
        case NETSOLVE_C:
            return sizeof(scomplex);
        case NETSOLVE_Z:
            return sizeof(dcomplex);
        case NETSOLVE_CHAR:
            return sizeof(char);
        case NETSOLVE_B:
            return sizeof(char);
        default:
            ns_errno = NetSolveInternalError;
            return -1;
    }
}



/*************************************************************************
    netsolve_xdrsizeof
    Returns the size in bytes of Netsolve data types once they are
    XDR encoded.
*************************************************************************/
int netsolve_xdrsizeof(int data_type)
{
    switch(data_type) {
        case NETSOLVE_I:
            return XDR_SIZEOF_INT;
        case NETSOLVE_D:
            return XDR_SIZEOF_DOUBLE;
        case NETSOLVE_S:
            return XDR_SIZEOF_FLOAT;
        case NETSOLVE_C:
            return XDR_SIZEOF_SCOMPLEX;
        case NETSOLVE_Z:
            return XDR_SIZEOF_DCOMPLEX;
        case NETSOLVE_CHAR:
            return XDR_SIZEOF_CHAR;
        case NETSOLVE_B:
            return sizeof(char);
        default:
            ns_errno = NetSolveInternalError;
            return -1;
    }
}
```

```
/*************************************************************************
    sendArray
    Send an array of typed data on the current communicator.
*************************************************************************/
int sendArray(Communicator *comm,int data_type,void *data,int nb)
{
    XDR xdr_stream;
    char *buffer = NULL;
    void *tosend = NULL;
    int size;
    int cc=1;

    if (comm->encoding == DATA_XDR)
        size = nb*netsolve_xdrsizeof(data_type);
    else
        size = nb*netsolve_sizeof(data_type);

    if (comm->encoding == DATA_XDR) {
        buffer = (char *)calloc(size,sizeof(char));
        xdrmem_create(&xdr_stream,buffer,size,XDR_FREE);
        xdrmem_create(&xdr_stream,buffer,size,XDR_ENCODE);

        switch(data_type) {
        case NETSOLVE_I:
            if (xdr_vector(&xdr_stream,data,nb,sizeof(int),xdr_int) != 1)
                cc =    -1;
            tosend = buffer;
            break;
        case NETSOLVE_S:
            if (xdr_vector(&xdr_stream,data,nb,sizeof(float),xdr_float) != 1)
                cc =    -1;
             tosend = buffer;
             break;
        case NETSOLVE_D:
            if (xdr_vector(&xdr_stream,data,nb,sizeof(double),xdr_dbuble) != 1)
                cc = -1;
            tosend = buffer;
            break;
        case NETSOLVE_CHAR:
            if (xdr_vector(&xdr_stream,data,nb,sizeof(char),xdr_char) != 1)
                cc = -1;
            tosend = buffer;
            break;
        case NETSOLVE_C:
            if (xdr_vector(&xdr_stream,data,nb,sizeof(scomplex),xdr_scomplex) != 1)
                cc = -1;
            tosend = buffer;
            break;
        case NETSOLVE_Z:
            if (xdr_vector(&xdr_stream,data,nb,sizeof(dcomplex),xdr_dcomplex) != 1)
                cc = -1;
            tosend = buffer;
            break;
        case NETSOLVE_B: /* No XDR encoding there */
            tosend = data;
            break;
        }
```

```
    }
    else {
        tosend = data;
    }

    if (cc == -1) {
        if (comm->encoding == DATA_XDR) {
            free(buffer);
            xdr_destroy(&xdr_stream);
        }
        ns_errno = NetSolveNetworkError;
        return -1;
    }

    cc = SendData(comm->sock, tosend, size);

    if (comm->encoding == DATA_XDR) {
        free(buffer);
        xdr_destroy(&xdr_stream);
    }
    if (cc != size) {
        ns_errno = NetSolveNetworkError;
        return -1;
    }
    else
        return nb;
}

/*************************************************************************
    recvArray
    Receive an array of typed data on the current communicator.
*************************************************************************/
int recvArray(Communicator *comm, int data_type, void *data, int nb)
{
    XDR xdr_stream;
    char *buffer = NULL;
    int size;
    int cc;
    char *torecv;

    if (comm->encoding == DATA_XDR)
        size = nb*netsolve_xdrsizeof(data_type);
    else
        size = nb*netsolve_sizeof(data_type);

    if (comm->encoding == DATA_XDR) {
        switch(data_type) {
            case NETSOLVE_B:
                torecv = data;
                break;
            default:
                buffer = (char *)calloc(size,sizeof(char));
                torecv = buffer;
                break;
        }
    }
    else {
```

```
        torecv = data;
    }

    cc = ReceiveData(comm->sock,torecv,size);
    if (cc == SOCKET_ERROR) {
        if (buffer)
            free(buffer);
        ns_errno = NetSolveNetworkError;
        return -1;
    }

    if (comm->encoding == DATA_XDR) {
        xdrmem_create(&xdr_stream,buffer,size,XDR_FREE);
        xdrmem_create(&xdr_stream,buffer,size,XDR_DECODE);

        switch(data_type) {
        case NETSOLVE_I:
            if (xdr_vector(&xdr_stream,data,nb,sizeof(int),xdr_int) != 1)
                cc = -1;
            break;
        case NETSOLVE_S:
            if (xdr_vector(&xdr_stream,data,nb,sizeof(float),xdr_float) != 1)
                cc = -1;
            break;
        case NETSOLVE_D:
            if (xdr_vector(&xdr_stream,data,nb,sizeof(double),xdr_double) != 1)
                cc = -1;
            break;
        case NETSOLVE_CHAR:
            if (xdr_vector(&xdr_stream,data,nb,sizeof(char),xdr_char) != 1)
                cc = -1;
            break;
        case NETSOLVE_C:
            if (xdr_vector(&xdr_stream,data,nb,sizeof(scomplex),xdr_scomplex) != 1)
                cc = -1;
            break;
        case NETSOLVE_Z:
            if (xdr_vector(&xdr_stream,data,nb,sizeof(dcomplex),xdr_dcomplex) != 1)
                cc = -1;
            break;
        case NETSOLVE_B: /* No XDR encoding there */
            /* Nothing */
            break;
        }
    }

    if (comm->encoding == DATA_XDR) {
        free(buffer);
        xdr_destroy(&xdr_stream);
    }
    if (cc == size)
        return nb;
    else {
        ns_errno = NetSolveNetworkError;
        return -1;
    }
}
```

```
/************************************************************************
    sendInt
    Sends an integer (to make the code look nicer).
************************************************************************/
int sendInt(Communicator *comm,int *ptr)
{
    return sendArray(comm,NETSOLVE_I,ptr,1);
}



/************************************************************************
    recvInt
    Receives an integer (to make the code look nicer).
************************************************************************/
int recvInt(Communicator *comm,int *ptr)
{
    return recvArray(comm,NETSOLVE_I,ptr,1);
}



/************************************************************************
    sendString
    Send a string, prepended with its length.
    Returns the number of bytes sent.
************************************************************************/
int sendString(Communicator *comm,char *s)
{
    int size;
    char *tosend;

    if (s == NULL)
        tosend = strdup("");
    else
        tosend = s;

    size = strlen(tosend)+1;

    if (sendInt(comm,&size) != 1) {
        ns_errno = NetSolveNetworkError;
        return -1;
    }
    if (sendArray(comm,NETSOLVE_CHAR,tosend,size) != size) {
        ns_errno = NetSolveNetworkError;
        return -1;
    }
    if (s == NULL)
        free(tosend);
    return size;
}
```

```
/************************************************************************
    recvString
    Receives a string, and allocates the space for it !!
    Returnes the number of bytes received.
************************************************************************/
int recvString(Communicator *comm, char **s)
{
    int size;

    if (recvInt(comm,&size) != 1) {
        ns_errno = NetSolveNetworkError;
        return -1;
    }
    *s = (char *)calloc(size,sizeof(char));
    if (recvArray(comm,NETSOLVE_CHAR,*s,size) != size) {
        free(*s);
        *s = NULL;
        return -1;
    }
    return size;
}



/************************************************************************
    sendIPaddr
    Sends an IPaddr. Takes care of the XDR screw up.
************************************************************************/
int sendIPaddr(Communicator *comm,unsigned int *IPaddr)
{
    XDR xdr_stream;
    char *buffer = NULL;
    int size;
    void *tosend;
    int cc=0;

    if (comm->encoding == DATA_XDR) {
        size = 4*XDR_SIZEOF_U_CHAR;
        buffer = (char *)calloc(size,sizeof(char));
        xdrmem_create(&xdr_stream,buffer,size,XDR_FREE);
        xdrmem_create(&xdr_stream,buffer,size,XDR_ENCODE);
#ifndef PIAT_T3E
        if (xdr_u_char(&xdr_stream,&(((unsigned char *)IPaddr)[0])) != 1)
            cc = -1;
        if (xdr_u_char(&xdr_stream,&(((unsigned char *)IPaddr)[1])) != 1)
            cc = -1;
        if (xdr_u_char(&xdr_stream,&(((unsigned char *)IPaddr)[2])) != 1)
            cc = -1;
        if (xdr_u_char(&xdr_stream,&(((unsigned char *)IPaddr)[3])) != 1)
            cc = -1;
#else
        if (xdr_u_char(&xdr_stream,&(((char *)IPaddr)[0])) != 1)
            cc = -1;
        if (xdr_u_char(&xdr_stream,&(((char *)IPaddr)[1])) != 1)
            cc = -1;
        if (xdr_u_char(&xdr_stream,&(((char *)IPaddr)[2])) != 1)
            cc = -1;
```

```
        if (xdr_u_char(&xdr_stream,&(((char *)IPaddr)[3])) != 1)
            cc = -1;
#endif
        tosend = buffer;
    }
    else {
        size = sizeof(unsigned int);
        tosend = IPaddr;
    }

    if (cc == -1) {
        if (comm->encoding == DATA_XDR) {
            free(buffer);
            xdr_destroy(&xdr_stream);
        }
        ns_errno = NetSolveNetworkError;
        return -1;
    }

    cc = SendData(comm->sock,tosend,size);
    if (comm->encoding == DATA_XDR) {
        free(buffer);
        xdr_destroy(&xdr_stream);
    }
    if (cc == size)
        return 1;
    else {
        ns_errno = NetSolveNetworkError;
        return -1;
    }
}



/*************************************************************************
    recvIPaddr
    Receives an IPaddr. Takes care of the XDR screw up.
*************************************************************************/
int recvIPaddr(Communicator *comm,unsigned int *IPaddr)
{
    XDR xdr_stream;
    char *buffer = NULL;
    int size;
    void *torecv;
    int cc;


    if (comm->encoding == DATA_XDR) {
        size = 4*XDR_SIZEOF_U_CHAR;
        buffer = (char *)calloc(size,sizeof(char));
        torecv = buffer;
    }
    else {
        size = sizeof(unsigned int);
        torecv = IPaddr;
    }
```

```
    cc = ReceiveData(comm->sock,torecv,size);
    if (cc == SOCKET_ERROR) {
        if (buffer)
            free(buffer);
        ns_errno = NetSolveNetworkError;
        return -1;
    }

    if (comm->encoding == DATA_XDR) {
        xdrmem_create(&xdr_stream,buffer,size,XDR_FREE);
        xdrmem_create(&xdr_stream,buffer,size,XDR_DECODE);
#ifndef PIAT_T3E
        if (xdr_u_char(&xdr_stream,&(((unsigned char *)IPaddr)[0])) != 1)
            cc = -1;
        if (xdr_u_char(&xdr_stream,&(((unsigned char *)IPaddr)[1])) != 1)
            cc = -1;
        if (xdr_u_char(&xdr_stream,&(((unsigned char *)IPaddr)[2])) != 1)
            cc = -1;
        if (xdr_u_char(&xdr_stream,&(((unsigned char *)IPaddr)[3])) != 1)
            cc = -1;
#else
        if (xdr_u_char(&xdr_stream,&(((char *)IPaddr)[0])) != 1)
            cc = -1;
        if (xdr_u_char(&xdr_stream,&(((char *)IPaddr)[1])) != 1)
            cc = -1;
        if (xdr_u_char(&xdr_stream,&(((char *)IPaddr)[2])) != 1)
            cc = -1;
        if (xdr_u_char(&xdr_stream,&(((char *)IPaddr)[3])) != 1)
            cc = -1;
#endif
    }

    if (comm->encoding == DATA_XDR) {
        free(buffer);
        xdr_destroy(&xdr_stream);
    }
    if (cc == -1) {
        ns_errno = NetSolveNetworkError;
        return -1;
    }
    else
        return 1;
}
```

```
/*************************************************************************
    sendArrayFromFile
    Sends an array from a file to the communicator.
*************************************************************************/
int sendArrayFromFile(Communicator *comm,int data_type,int fd,int nb)
{
    char buffer[BUFFSIZE];
    int count=nb;
    int maximum=BUFFSIZE/netsolve_sizeof(data_type);
    int loaded;

    while(count != 0) {
        /* Load data in the buffer from the file */
        loaded = MIN(maximum,count);
        if (read(fd,buffer,loaded*netsolve_sizeof(data_type)) !=
                        netsolve_sizeof(data_type)*loaded) {
            ns_errno = NetSolveFileError;
            return -1;
        }
        /* Send that on the communicator */
        if (sendArray(comm,data_type,buffer,loaded) != loaded)
            return -1;
        count-=loaded;
    }
    return nb;
}



/*************************************************************************
    recvArrayToFile
    Receives an arry from the communicator into a file.
*************************************************************************/
int recvArrayToFile(Communicator *comm,int data_type,int fd,int nb)
{
    char buffer[BUFFSIZE];
    int count=nb;
    int maximum=BUFFSIZE/netsolve_sizeof(data_type);
    int loaded;

    while(count != 0) {
        /* Receive data on the communicator */
        loaded = MIN(maximum,count);
        if (recvArray(comm,data_type,buffer,loaded) != loaded)
            return -1;
        /* Store that from the buffer into the file */
        if (write(fd,buffer,loaded*netsolve_sizeof(data_type)) !=
                        netsolve_sizeof(data_type)*loaded) {
            ns_errno = NetSolveFileError;
            return -1;
        }
        count-=loaded;
    }
    return nb;
}
```

```
/************************************************************************
    sendFileAsString
************************************************************************/
int sendFileAsString(Communicator *comm,char *filename)
{
    char *buffer;
    struct stat st;
    int fd;

    if (stat(filename,&st)) {
        ns_errno = NetSolveFileError;
        return -1;
    }

    fd = open(filename,O_RDONLY,0666);
    if (fd < 0) {
        ns_errno = NetSolveFileError;
        return -1;
    }

    buffer = (char *)calloc(st.st_size+1,sizeof(char));
    if (read(fd,buffer,st.st_size*sizeof(char)) != st.st_size*sizeof(char)) {
        free(buffer);
        close(fd);
        ns_errno = NetSolveFileError;
        return -1;
    }
    if (sendString(comm,buffer) == -1) {
        free(buffer);
        close(fd);
        return -1;
    }
    free(buffer);
    close(fd);
    return 1;
}
```

Appendix D: Source Code for clientutil.c

```
/*************************************************************************/
/*  clientutil.c                                                         */
/*                                                                       */
/*  Based on original work by Henri Casanova                             */
/*  Win32 version by Rick Phillips                                       */
/*************************************************************************/

#include "core.h"
#include "client.h"
#include <signal.h>
#include <time.h>


/*************************************************************************
    getNumberOfServers
*************************************************************************/
int getNumberOfServers(char *nickname)
{
    SOCKET sock;
    Communicator *comm;
    char *agent_name;
    int tag;
    int nb_servers;

    agent_name = getNetSolveAgent();
    if (agent_name == NULL) {
        ns_errno = NetSolveSetNetSolveAgent;
        return -1;
    }

    sock = connectToSocket(agent_name,0,AGENT_PORT);
    if (sock == SOCKET_ERROR)
        return -1;

    comm = initTransaction(sock,DATA_XDR);
    if (comm == NULL) {
        closesocket(sock);
        return -1;
    }

    tag = NS_PROT_NB_SERVERS;
    if (sendInt(comm,&tag) == -1) {
        endTransaction(comm);
        return -1;
    }
    if (sendString(comm,nickname) == -1) {
        endTransaction(comm);
        return -1;
    }
    if (recvInt(comm,&nb_servers) == -1) {
        endTransaction(comm);
        return -1;
    }
    endTransaction(comm);
```

```
    ns_errno = NetSolveOK;
    return nb_servers;
}



/*************************************************************************
    determineSizes
*************************************************************************/
int determineSizes(ProblemDesc *pd,Object **input,Object **output,
                   int *input_size,int *output_size, int *problem_size)
{
    int i;
    int tmp;

    *input_size = 0;
    for (i=0;i<pd->nb_input_objects;i++) {
        tmp = getObjectByteSize(input[ i] );
        if (tmp == -1)
            return -1;
        (*input_size) += tmp;
    }

    *output_size = 0;
    for (i=0;i<pd->nb_output_objects;i++) {
        tmp = getObjectByteSize(output[ i] );
        if (tmp == -1)
            return -1;
        (*output_size) += tmp;
    }

    *problem_size = 1;
    for (i=0;i<pd->nb_input_objects;i++) {
         if (input[ i]->object_type == NETSOLVE_MATRIX) {
            *problem_size = MAX(*problem_size,input[ i]->attributes.matrix_attributes.m);
            *problem_size = MAX(*problem_size,input[ i]->attributes.matrix_attributes.n);
         }
    }
    if (*problem_size == 1) {
        for (i=0;i<pd->nb_input_objects;i++) {
            if (input[ i]->object_type == NETSOLVE_VECTOR)
                *problem_size = MAX(*problem_size,input[ i]-
>attributes.vector_attributes.m);
        }
    }
    ns_errno = NetSolveOK;
    return 1;
}
```

```
/*************************************************************************
    submitProblemToAgent
*************************************************************************/
int submitProblemToAgent(char *nickname,int input_size, int output_size,
                         int problem_size, int *nb_servers, char ***hostlist,
                         unsigned int **IPaddrlist, int **portlist,
                         int **dataformatlist, int **predictedtimelist)
{
    SOCKET sock;
    Communicator *comm;
    char *agent_name;
    int tag;
    int i;

    agent_name = getNetSolveAgent();
    if (agent_name == NULL) {
        ns_errno = NetSolveSetNetSolveAgent;
        return -1;
    }

    sock = connectToSocket(agent_name,0,AGENT_PORT);
    if (sock == SOCKET_ERROR)
        return -1;

    comm = initTransaction(sock,DATA_XDR);
    if (comm == NULL) {
        closesocket(sock);
        return -1;
    }

    /* Sending */
    tag = NS_PROT_PROBLEM_SUBMIT;
    if ((sendInt(comm,&tag) == -1) ||
        (sendString(comm,nickname) == -1) ||
        (sendInt(comm,&input_size) == -1) ||
        (sendInt(comm,&output_size) == -1) ||
        (sendInt(comm,&problem_size) == -1)) {
        endTransaction(comm);
        return -1;
    }

    /* receiving */
    if (recvInt(comm,&tag) == -1) {
        endTransaction(comm);
        return -1;
    }
    if (tag == NS_PROT_PROBLEM_NOT_FOUND) {
        endTransaction(comm);
        ns_errno = NetSolveUnknownProblem;
        return -1;
    }
    if (tag != NS_PROT_OK) {
        endTransaction(comm);
        ns_errno = NetSolveProtocolError;
        return -1;
    }
```

```
    if (recvInt(comm,nb_servers) == -1) {
        endTransaction(comm);
        return -1;
    }
    if (*nb_servers == 0) {
        ns_errno = NetSolveNoServer;
        return -1;
    }

    *hostlist = (char **)calloc(*nb_servers,sizeof(char*));
    *IPaddrlist = (unsigned int *)calloc(*nb_servers,sizeof(unsigned int));
    *portlist = (int *)calloc(*nb_servers,sizeof(int));
    *dataformatlist  = (int *)calloc(*nb_servers,sizeof(int));
    *predictedtimelist = (int *)calloc(*nb_servers,sizeof(int));

    for (i=0;i<*nb_servers;i++) {
        if ((recvString(comm,&((*hostlist)[i])) == -1) ||
            (recvIPaddr(comm,&((*IPaddrlist)[i])) == -1) ||
            (recvInt(comm,&((*portlist)[i])) == -1) ||
            (recvInt(comm,&((*dataformatlist)[i])) == -1) ||
            (recvInt(comm,&((*predictedtimelist)[i])) == -1)) {
           endTransaction(comm);
           return -1;
        }
    }
    endTransaction(comm);
    ns_errno = NetSolveOK;
    return 1;
}



/************************************************************************
    submitProblemToServer
************************************************************************/
int submitProblemToServer(RequestDesc *rd)
{
    int encoding;
    SOCKET sock;
    int tag,i;
    time_t now;
    char *agent_name;
    int ack;

    if (rd->server_data_format == getArch())
        encoding = DATA_RAW;
    else
        encoding = DATA_XDR;

#ifdef VIEW
    fprintf(stderr,"Contacting server on '%s'...\n",rd->server_hostname);
#endif
    sock = connectToSocket(rd->server_hostname, rd->server_IPaddr,  rd->server_port);
    if (sock == SOCKET_ERROR)
        return -1;

    rd->comm = initTransaction(sock,encoding);
```

```
if (rd->comm == NULL) {
    closesocket(sock);
    fprintf(stderr,"HERE\n");
    return -1;
}

/* Sending */
tag = NS_PROT_PROBLEM_SOLVE;

if (sendInt(rd->comm, &tag) == -1) {
    fprintf(stderr,"THERE\n");
    endTransaction(rd->comm);
    return -1;
}
if (sendProblemDesc(rd->comm,rd->pd) == -1) {
    endTransaction(rd->comm);
    return -1;
}

/* receiving */
if (recvInt(rd->comm, &tag) == -1) {
    endTransaction(rd->comm);
    return -1;
}
if (tag == NS_PROT_PROBLEM_NOT_FOUND) {
    endTransaction(rd->comm);
    ns_errno = NetSolveUnknownProblem;
    return -1;
}
if (tag == NS_PROT_BAD_SPECIFICATION) {
    endTransaction(rd->comm);
    ns_errno = NetSolveBadProblemSpecification;
    return -1;
}
if (tag == NS_PROT_NOT_ALLOWED) {
    endTransaction(rd->comm);
    ns_errno = NetSolveNotAllowed;
    return -1;
}
if (tag != NS_PROT_ACCEPTED) {
    endTransaction(rd->comm);
    ns_errno = NetSolveProtocolError;
    return -1;
}

/* sending my major */
if (sendInt(rd->comm, &my_major) == -1) {
    endTransaction(rd->comm);
    return -1;
}
/* sending the agent name */
agent_name = getNetSolveAgent();
if (sendString(rd->comm, agent_name) == -1) {
    free(agent_name);
    endTransaction(rd->comm);
    return -1;
}
```

```
    /* sending the objects */
    for (i=0;i<rd->pd->nb_input_objects;i++) {
        if (sendObject(rd->comm,rd->input[i]) == -1) {
            endTransaction(rd->comm);
            return -1;
        }
    }
    time(&now);
    rd->start_time = (int)now;

    /* Get the ack from the server */
    if (recvInt(rd->comm,&ack) == -1) {
        endTransaction(rd->comm);
        return -1;
    }
    if (ack == NS_PROT_SERVICE_PID) {
        if (recvInt(rd->comm,&(rd->pid)) == -1) {
            endTransaction(rd->comm);
            return -1;
        }
        ns_errno = NetSolveOK;
        return 1;
    }
    else {
        switch(ack) {
            case NS_PROT_INTERNAL_FAILURE:
                ns_errno = NetSolveInternalError;
                break;
            case NS_PROT_BAD_VALUES:
                ns_errno = NetSolveBadValues;
                break;
            case NS_PROT_DIM_MISMATCH:
                ns_errno = NetSolveDimensionMismatch;
                break;
            case NS_PROT_NO_SOLUTION:
                ns_errno = NetSolveNoSolution;
                break;
            case NS_PROT_UPF_ERROR:
                ns_errno = NetSolveUPFError;
                break;
            case NS_PROT_UPF_UNSAFE:
                ns_errno = NetSolveUPFUnsafe;
                break;
            default:
                ns_errno = NetSolveInternalError;
                break;
        }
        return -1;
    }
    return -1;
}
```

```
/**************************************************************************
    retrieveResultFromServerBlockNoBlock
**************************************************************************/
int retrieveResultFromServerBlockNoBlock(int block_noblock,RequestDesc *rd)
{
    int tag;
    int i;
    char *stdoutbuf;
    int hbm_status;
    int ready;

    if (block_noblock == NOBLOCK) {
        ready = isThereSomethingOnTheSocket(rd->comm);
        if (ready == -1) {
            endTransaction(rd->comm);
            if (rd->hbmcomm != NULL) {
                endTransaction(rd->hbmcomm);
                rd->hbmcomm = NULL;
            }
            return -1;
        }
        if (ready == 0) {
            if (rd->hbmactivated != -1)
                hbm_status = getStatusFromHBM(rd);
            else
                hbm_status = -1;

            if (hbm_status == -1)
                rd->hbmactivated = -1;
            else {
                switch(hbm_status) {
                    case 1:
                    case 2:
                    case 3:
                    case 4:
                        break;
                    default:
                        endTransaction(rd->comm);
                        endTransaction(rd->hbmcomm);
                        rd->hbmcomm = NULL;
                        ns_errno = NetSolveNetworkError;
                        return -1;
                }
            }
            ns_errno = NetSolveNotReady;
            return -1;
        }
    }
    else {
        while(1) {
            ready = isThereSomethingOnTheSocket(rd->comm);
            if (ready == -1) {
                endTransaction(rd->comm);
                if (rd->hbmcomm != NULL) {
                    endTransaction(rd->hbmcomm);
                    rd->hbmcomm = NULL;
                }
```

```
            return -1;
        }
        if (ready != 0)
            break;
        /* At this point, still pending...*/
        if (rd->hbmactivated != -1) {
            hbm_status = getStatusFromHBM(rd);
            if (hbm_status == -1) {
                rd->hbmactivated = -1;
            }
            else {
                switch(hbm_status) {
                    case 1:
                    case 2:
                    case 3:
                    case 4:
                        break;  /* just continue in the while loop */
                    default:
                        endTransaction(rd->comm);
                        endTransaction(rd->hbmcomm);
                        rd->hbmcomm = NULL;
                        ns_errno = NetSolveNetworkError;
                        return -1;
                }
            }
        }
        sleep(3);
    }
}

/* At this point, kill whatever connection was left with the HBM */
if (rd->hbmcomm != NULL) {
    endTransaction(rd->hbmcomm);
    rd->hbmcomm = NULL;
}

/* Receive the stdout */
if (recvString(rd->comm,&stdoutbuf) == -1) {
    endTransaction(rd->comm);
    return -1;
}
fprintf(STDOUT stdoutbuf);
free(stdoutbuf);

if (recvInt(rd->comm,&tag) == -1) {
    endTransaction(rd->comm);
    return -1;
}

if (tag != NS_PROT_SOLVED) {
    endTransaction(rd->comm);
    switch(tag) {
        case NS_PROT_INTERNAL_FAILURE:
            ns_errno = NetSolveServerError;
        case NS_PROT_BAD_VALUES:
            ns_errno = NetSolveBadValues;
```

```
            case NS_PROT_DIM_MISMATCH:
                ns_errno = NetSolveDimensionMismatch;
            case NS_PROT_NO_SOLUTION:
                ns_errno = NetSolveNoSolution;
            default:
                ns_errno = NetSolveProtocolError;
        }
        return -1;
    }

    for (i=0;i<rd->pd->nb_output_objects;i++) {
        if (recvObject(rd->comm,rd->output[ i]) == -1) {
            endTransaction(rd->comm);
            return -1;
        }
    }
    endTransaction(rd->comm);
    reportJobCompleted(rd->server_IPaddr);
    ns_errno = NetSolveOK;
    return 1;
}



/*************************************************************************
    reportJobCompleted
*************************************************************************/
int reportJobCompleted(unsigned int IPaddr)
{
    SOCKET sock;
    Communicator *comm;
    char *agent_name;
    int tag;

    agent_name = getNetSolveAgent();
    if (agent_name == NULL) {
        ns_errno = NetSolveSetNetSolveAgent;
        return -1;
    }

    sock = connectToSocket(agent_name,0,AGENT_PORT);
    if (sock == SOCKET_ERROR)
        return -1;

    comm = initTransaction(sock,DATA_XDR);
    if (comm == NULL) {
        closesocket(sock);
        return -1;
    }

    /* Sending */
    tag = NS_PROT_JOB_COMPLETED;
    if (sendInt(comm,&tag) == -1) {
        endTransaction(comm);
        return -1;
    }
```

```
    if (sendIPaddr(comm,&IPaddr) == -1) {
        endTransaction(comm);
        return -1;
    }
    endTransaction(comm);

    ns_errno = NetSolveOK;
    return 1;
}


/************************************************************************
    reportFailure
************************************************************************/
int reportFailure(unsigned int IPaddr, char *nickname, int error_type)
{
    SOCKET sock;
    Communicator *comm;
    char *agent_name;
    int tag;

    agent_name = getNetSolveAgent();
    if (agent_name == NULL) {
        ns_errno = NetSolveSetNetSolveAgent;
        return -1;
    }

    sock = connectToSocket(agent_name,0,AGENT_PORT);
    if (sock == SOCKET_ERROR)
        return -1;

    comm = initTransaction(sock,DATA_XDR);
    if (comm == NULL) {
        closesocket(sock);
        return -1;
    }

    /* Sending */
    tag = NS_PROT_SV_FAILURE;
    if ((sendInt(comm,&tag) == -1) ||
            (sendIPaddr(comm,&IPaddr) == -1) ||
            (sendString(comm,nickname) == -1) ||
            (sendInt(comm,&error_type) == -1)) {
        endTransaction(comm);
        return -1;
    }
    endTransaction(comm);

    ns_errno = NetSolveOK;
    return 1;
}
```

```
/*****************************************************************************
    getAllProblems
*****************************************************************************/
int getAllProblems(ProblemDesc ***list,char *agent_name)
{
    SOCKET sock;
    Communicator *comm;
    int nb_problems;
    int i;
    int tag;

    sock = connectToSocket(agent_name,0,AGENT_PORT);
    if (sock == SOCKET_ERROR) {
        *list = NULL;
        return -1;
    }

    comm = initTransaction(sock,DATA_XDR);
    if (comm == NULL) {
        closesocket(sock);
        *list = NULL;
        return -1;
    }

    /* Sending */
    tag = NS_PROT_PROBLEM_LIST;
    if (sendInt(comm,&tag) == -1) {
        endTransaction(comm);
        *list = NULL;
        return -1;
    }

    /* receiving */
    if (recvInt(comm,&nb_problems) == -1) {
        endTransaction(comm);
        *list = NULL;
        return -1;
    }

    *list = (ProblemDesc **)calloc(nb_problems,sizeof(ProblemDesc*));
    for (i=0;i<nb_problems;i++) {
        (*list)[i] = recvProblemDesc(comm);
        if ((*list)[i] == NULL) {
            *list = NULL;
            return -1;
        }
    }
    endTransaction(comm);
    return nb_problems;
}
```

```
/*****************************************************************************
    getAllAgents
*****************************************************************************/
int getAllAgents(AgentDesc ***list,char *agent_name)
{
    SOCKET sock;
    Communicator *comm;
    int nb_agents;
    int i;
    int tag;

    sock = connectToSocket(agent_name,0,AGENT_PORT);
    if (sock == SOCKET_ERROR) {
        *list = NULL;
        return -1;
    }

    comm = initTransaction(sock,DATA_XDR);
    if (comm == NULL) {
        closesocket(sock);
        *list = NULL;
        return -1;
    }

    /* Sending */
    tag = NS_PROT_AGENT_LIST;
    if (sendInt(comm,&tag) == -1) {
        endTransaction(comm);
        *list = NULL;
        return -1;
    }

    /* receiving */
    if (recvInt(comm,&nb_agents) == -1) {
        endTransaction(comm);
        *list = NULL;
        return -1;
    }

    *list = (AgentDesc **)calloc(nb_agents,sizeof(AgentDesc*));
    for (i=0;i<nb_agents;i++) {
        (*list)[i] = recvAgentDesc(comm);
        if ((*list)[i] == NULL) {
            *list = NULL;
            return -1;
        }
    }
    endTransaction(comm);
    return nb_agents;
}
```

```
/*****************************************************************************
    getAllServers
*****************************************************************************/
int getAllServers(ServerDesc ***list,char *agent_name)
{
    SOCKET sock;
    Communicator *comm;
    int nb_servers;
    int i;
    int tag;

    sock = connectToSocket(agent_name,0,AGENT_PORT);
    if (sock == SOCKET_ERROR) {
        *list = NULL;
        return -1;
    }

    comm = initTransaction(sock,DATA_XDR);
    if (comm == NULL) {
        closesocket(sock);
        *list = NULL;
        return -1;
    }

    /* Sending */
    tag = NS_PROT_SERVER_LIST;
    if (sendInt(comm,&tag) == -1) {
        endTransaction(comm);
        *list = NULL;
        return -1;
    }

    /* receiving */
    if (recvInt(comm,&nb_servers) == -1) {
        endTransaction(comm);
        *list = NULL;
        return -1;
    }

    *list = (ServerDesc **)calloc(nb_servers,sizeof(ServerDesc*));
    for (i=0;i<nb_servers;i++) {
        (*list)[i] = recvServerDesc(comm);
        if ((*list)[i] == NULL) {
            *list = NULL;
            return -1;
        }
    }
    endTransaction(comm);
    return nb_servers;
}
```

```
/*****************************************************************************
    netsolveInfo
*****************************************************************************/
int netsolveInfo(char *nickname, ProblemDesc **pd, char *agent_name)
{
    SOCKET sock;
    int tag;
    Communicator *comm;

    sock = connectToSocket(agent_name, 0, AGENT_PORT);
    if (sock == SOCKET_ERROR) {
        *pd = NULL;
        return -1;
    }

    comm = initTransaction(sock, DATA_XDR);
    if (comm == NULL) {
        *pd = NULL;
        closesocket(sock);
        return -1;
    }

    /* Sending */
    tag = NS_PROT_PROBLEM_INFO;
    if (sendInt(comm, &tag) == -1) {
        endTransaction(comm);
        *pd = NULL;
        return -1;
    }
    if (sendString(comm, nickname) == -1) {
        endTransaction(comm);
        *pd = NULL;
        return -1;
    }

    /* Receiving */
    if (recvInt(comm, &tag) == -1) {
        endTransaction(comm);
        *pd = NULL;
        return -1;
    }
    if (tag == NS_PROT_PROBLEM_NOT_FOUND) {
        ns_errno = NetSolveUnknownProblem;
        endTransaction(comm);
        *pd = NULL;
        return -1;
    }
    if (tag != NS_PROT_PROBLEM_DESC) {
        ns_errno = NetSolveProtocolError;
        endTransaction(comm);
        *pd = NULL;
        return -1;
    }

    *pd = recvProblemDesc(comm);
    endTransaction(comm);
```

```
    if (*pd == NULL)
        return -1;

    ns_errno = NetSolveOK;
    return 1;
}



/**************************************************************************
    getStatusFromHBM
**************************************************************************/
int getStatusFromHBM(RequestDesc *rd)
{
    SOCKET sock;
    int status;
    char *agent_name;

    if (rd->hbmcomm == NULL) {
        agent_name = getNetSolveAgent();
        sock = connectToSocket(agent_name,0,HBM_PROXY_PORT);
        if (sock == -1)
            return -1;

        rd->hbmcomm = initTransaction(sock,DATA_XDR);
        if (rd->hbmcomm == NULL)
            return -1;

        if (sendInt(rd->hbmcomm,&(rd->pid)) == -1) {
            endTransaction(rd->hbmcomm);
            rd->hbmcomm = NULL;
            return -1;
        }
    }

    if (recvInt(rd->hbmcomm,&status) == -1) {
        endTransaction(rd->hbmcomm);
        rd->hbmcomm = NULL;
        return -1;
    }
    while (isThereSomethingOnTheSocket(rd->hbmcomm)) {
        if (recvInt(rd->hbmcomm,&status) == -1) {
            endTransaction(rd->hbmcomm);
            rd->hbmcomm = NULL;
            return -1;
        }
    }
    return status;
}
```

Appendix E: Source Code for socketutil.c

```
/*************************************************************************/
/*  socketutil.c                                                         */
/*                                                                       */
/*  Based on original work by Henri Casanova                             */
/*  Win32 version by Rick Phillips                                       */
/*************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include <errno.h>
#include "core.h"
#include "win_util.h"

#define HOSTNAME_LENGTH 256



/*************************************************************************
    connectToSocket
*************************************************************************/
SOCKET connectToSocket(char *hostname,unsigned int IPaddr,int port)
{
    SOCKET sock;
    FAR struct sockaddr_in inet_addr;
    LPHOSTENT hp = NULL;

    /* Allocate a socket */
    if ((sock = socket(AF_INET,SOCK_STREAM,0)) == INVALID_SOCKET) {
        ns_errno = NetSolveSystemError;
        return SOCKET_ERROR;
    }
    BZERO((void*)&inet_addr,sizeof(inet_addr));

    /* Try to do a gethostbyaddr() */
    if (IPaddr)
        hp = gethostbyaddr((char *)&IPaddr,sizeof(unsigned int),AF_INET);
    if (hp == NULL) {
        /* Try to do a gethostbyname() */
        hp = gethostbyname(hostname);
        if (hp == NULL) {
            closesocket(sock);
            ns_errno = NetSolveUnknownHost;
            return SOCKET_ERROR;
        }
    }
    BCOPY((void*)hp->h_addr,(void*)&(inet_addr.sin_addr),hp->h_length);
    inet_addr.sin_family = AF_INET;
    inet_addr.sin_port = htons((short)port);
```

```
#ifdef DEBUG
    fprintf(STDERR"Connecting to %s ...\n",hostname);
#endif
    if (connect(sock,(struct sockaddr *)&inet_addr,sizeof(inet_addr)) == SOCKET_ERROR) {
        closesocket(sock);
        ns_errno = NetSolveConnectionRefused;
        return SOCKET_ERROR;
    }
    else
        return sock;
}



/*****************************************************************************
    isThereSomethingOnTheSocket
    returns :  0    if nothing on the socket
               1    something on the socket
              -1    if error and sets the errno
*****************************************************************************/
int isThereSomethingOnTheSocket(Communicator *comm)
{
    SOCKET sock = comm->sock;
    int off =0;
    int on  =1;
    char buf[ 1] ;
    int try;

    ioctlsocket(sock,FIONBIO,&on);
    try = recv(sock,buf,1,MSG_PEEK);
    ioctlsocket(sock,FIONBIO,&off);
    if (try == SOCKET_ERROR)
        return 0;
    if (try == 0) {
        ns_errno = NetSolveNetworkError;
        return -1;
    }
    ns_errno = NetSolveOK;
    return 1;
}
```

Vita

Rick Phillips was born in Knoxville, Tennessee on June 19, 1961. He and his family moved to Seymour, Tennessee in 1968. Then moved again to rural Sevier county, Tennessee in 1970. His mother, Ruth Phillips, and sister, Jan Phillips, still live there today. His father, Harry Phillips, lived there until June 4, 1998 when he passed away.

Rick graduated from Sevier County High School in 1979. Then, in 1984, he graduated with a Bachelor of Science degree in Computer Science from Tennessee Technological University. Most recently, Rick graduated with a Master of Science degree in Computer Science from the University of Tennessee at Knoxville.

While working on his Bachelor's degree, Rick worked as a co-op student for Martin Marietta in Oak Ridge, Tennessee. After receiving his Bachelor's degree in 1984, Rick accepted a full-time position with Martin Marietta. Martin Marietta later merged with Lockheed to form Lockheed Martin Corporation. Currently, Rick is employed as a Computing Specialist with Lockheed Martin Energy Research at the Oak Ridge National Laboratory.

Rick's professional interests include software development, computer networking, graphical user interfaces, and teaching.