NetSolve version 1.2: Design and Implementation

Henri Casanova [*] Jack Dongarra[*] [†]

November 6, 1998

Abstract

The design and implementation of NetSolve have been largely modified and improved in version 1.2. This document reviews the general architecture of the software, and gives many details about its implementation. This document is of interest to future NetSolve developers, to individuals who want to add on to NetSolve (e.g. new user interfaces), or to curious users.

[*]Department of Computer Science, University of Tennessee, TN 37996, USA
[†]Mathematical Science Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA

Contents

1 Introduction

This document is intended to provide information about the internals of NetSolve version 1.2. The NetSolve project started in the Summer of 1995. The first public release of an alpha version (1.0) occurred in January 1996 and generated a lot of feedback (suggestions, bug reports, new applications, ...). That feedback led to the release of NetSolve version 1.1.b in January 1998. NetSolve's popularity has been growing and the tools for building a *computational Grid* [1] have become more available. The 1.1.b design started to show its weaknesses in two ways: (i) adding new features needed by new users became problematic because of inappropriate design decisions; (ii) the seamless integration of new tools for the computation grid seemed difficult. Those observations motivated a complete rewrite and re-design of the software: NetSolve 1.2. The difference between NetSolve 1.1.b and NetSolve 1.2 will not be as striking to the NetSolve user (even though a number of new features and capabilities have been added) as to NetSolve developers. Like version 1.1.b, NetSolve 1.2 has been ported to most UNIX platforms. In addition, it provides Windows 95/NT C, Matlab and Mathematica client interfaces.

This document is organized as follows. Section 2 describes the general architecture of the software. Section 3 describes how networking is done in NetSolve 1.2. Section 4 lists the fundamental data structures. Section 5 details the protocols between agents, servers and clients. Sections 6.1, 6.2, 6.3, 6.4, and 6.5 describe general idea behind the implementation of the NetSolve client and its C, Fortran, Matlab, Mathematica, and Java APIs. Sections 7 and 8 gives information about the implementation of the NetSolve agent and server. Section 9 concludes the document with a set of ideas for short-term and long-term evolutions and improvements.

One of the difficulties about writing a description of the implementation of an ever-evolving research project is that detailed information becomes out-of-date rather quickly. We believe that this document is low-level enough to be relevant for future developers while being high-level enough so that it can be easily updated for future versions of the software. This is accomplished in several ways. First, this document shows NetSolve as a set of somewhat independent modules or subsystems (e.g. the *networking subsystem* in Section 3) and how each one of them can be entirely replaced by another subsystem of equivalent functionality. We expect this to happen more and more as grid-enabled tools become further stable and available. Second, this document contains a lot of hints and information that were gathered during the development of NetSolve. Those are mostly of general interest to readers with little experience with *portable* UNIX system programming and will be of use for future versions of NetSolve. Third, this document is structured such that it can be modified/upgraded easily when new versions of the software become available. Our goal is to make this document the implementation reference and to update it with any relevant modifications in the software.

References to NetSolve include numerous reports and publications [2, 3, 4, 5] as well as the latest edition of the Users' Guide [6]. We assume that the reader is familiar with the material in the Users' Guide.

2 Software Architecture

2.1 Overview

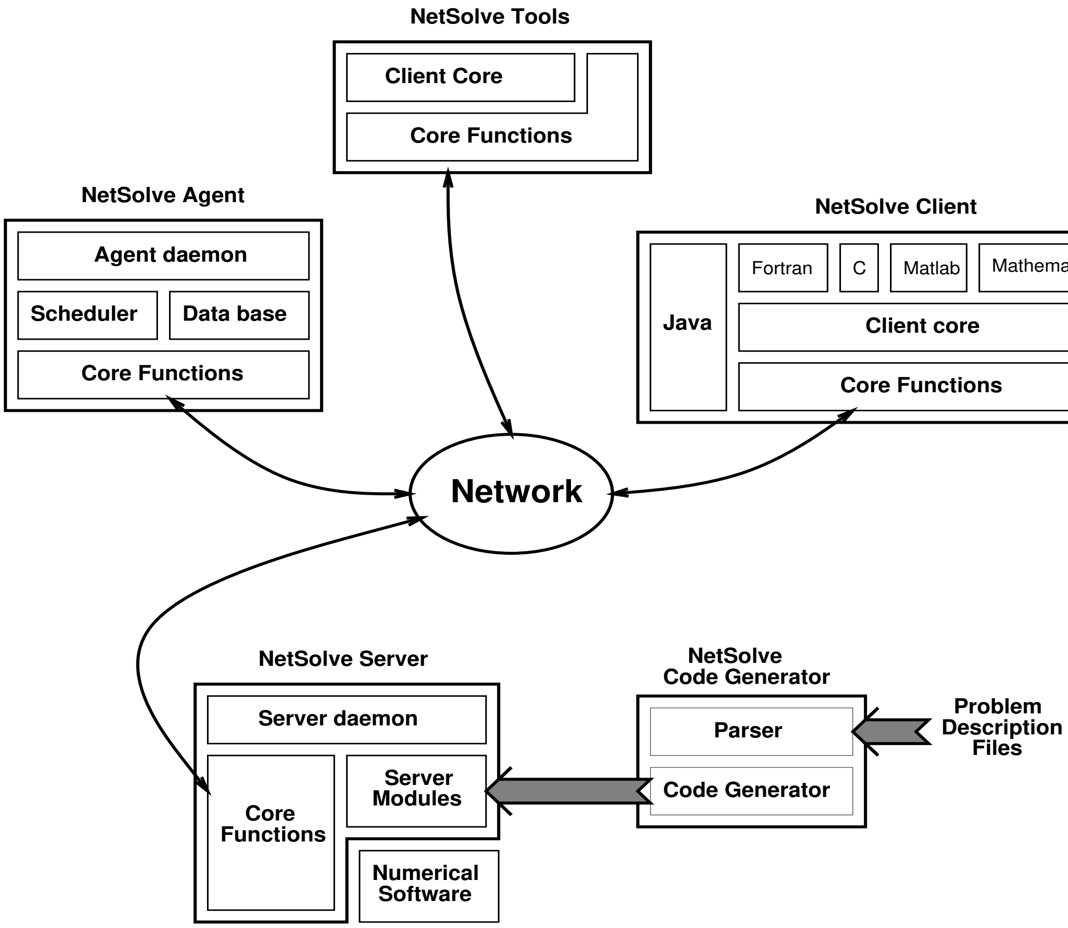

Figure 1: Software architecture

Figure 1 show the basic organization of the NetSolve software. There are 5 distinct components:

- The Agent,
- The Server,
- The Client,
- The Code Generator,
- The Tools.

The NetSolve agent and servers are detailed in Sections 7 and 8. The NetSolve client contained several interfaces detailed in Sections 6.2 to 6.5. All but the Java interface are build on top of a common set of routines called the *Client Core*. These routines implement basic client functionalities and are described in Section 6.1. The Code Generator is described in Section 8.2. The NetSolve tools are described in the Users' Guide and their implementation is rather trivial and will not be described in this document. Almost every component of the system is build on top of *Core Functions*. Those functions implement all the low-level functionalities in NetSolve: those that handle networking and manage the basic data structures (see Sections 3 and 4). Finally, the protocol used by modules to exchange information over the network is described in Section 5.

2.2 Compilation

Even though the compilation procedure is most likely to undergo changes (e.g. use of `autoconf`), we still deem it necessary to say a few words about it. In NetSolve version 1.2 the compilation in done with `make` which is a somewhat portable way of compiling software. However, experience shows that only a small subset of its functionalities is truly portable. In fact, [7] says: "... many useful features have been added by various implementors after `make` had time to spread and to develop into different variants...their use definitely reduces the portability of your description files". And furthermore, "many programmers have added features to `make` without updating the documentation". The rule of thumb that we recommend is: *if a feature seems unusually useful, it is probably not portable.*

The main makefile is located `$NETSOLVE_ROOT/src`. From now on we will assume that the current directory is `$NETSOLVE_ROOT` and we will denote subdirectories as `./src`. `./src/Makefile` calls and includes a number of other makefiles. Some of those makefiles are generated at compile time by the Code Generator (see Section 8.2). Figure 2 shows the entire makefile structure with all the makefiles:

- `./src/Makefile`: main entry-point,
- `./conf/conf.def`: general settings,
- `./conf/$NETSOLVE_ARCH.def`: machine dependent settings,
- `./src/Makefile.def`: general variable definitions,
- `./src/Makefile.object`: object rules,
- `./src/Makefile.numerical`: computational module makefile,
- `./src/Makefile.num_libs`: numerical software dependencies,
- `./src/Makefile.sample_software`: sample software makefile.

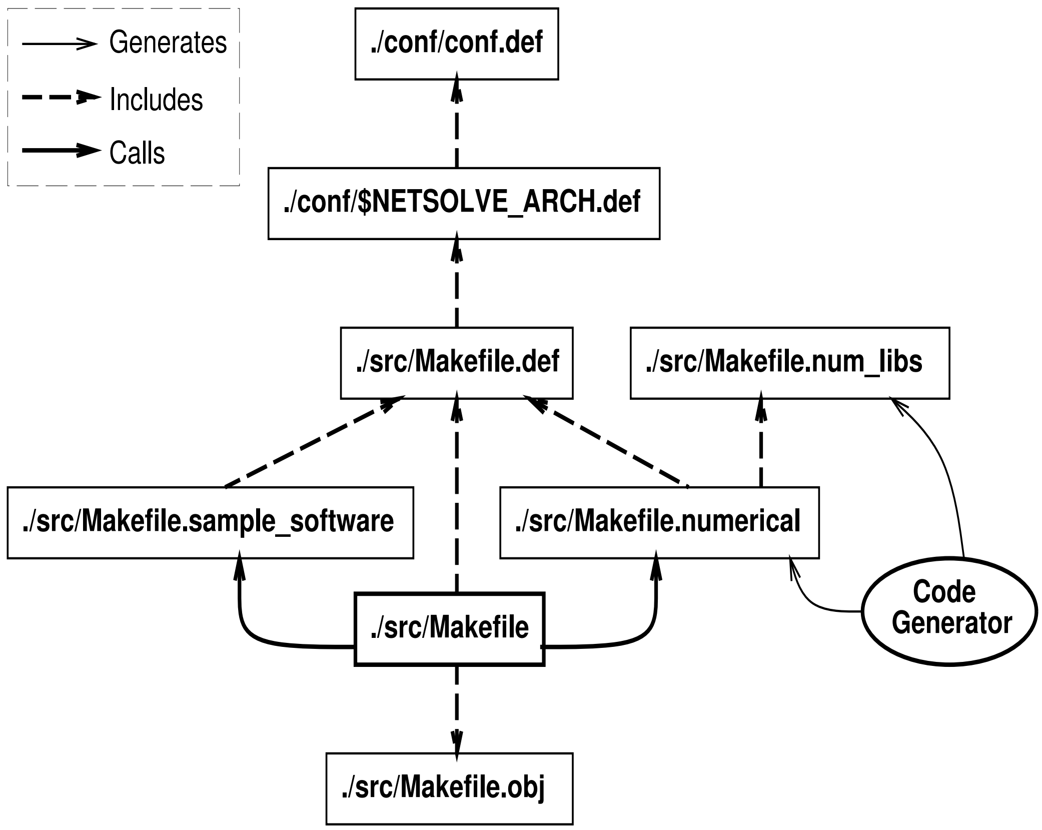

Figure 2: Makefile structure

2.3 Source Code

In this section, we give a list of all the source code directories in the current NetSolve distribution:

- `./src/Agent`: The agent,
- `./src/CFortran`: The C and Fortran APIs,
- `./src/ClientCore`: The client core functions,
- `./src/CodeGenerator`: The code generator,
- `./src/CoreFunctions`: The core functions,
- `./src/Demo`: The demos,
- `./src/Examples`: The C, Fortran and farming examples,
- `./src/Farming`: The farming interface,
- `./src/GlobusHBMWrappers`: Wrappers around the Globus Heart Beat Monitor,

- `./src/MCellInterface`: The interface to MCell (see [8]),
- `./src/Mathematica`: The Mathematica interface,
- `./src/Matlab`: The Matlab interface,
- `./src/SampleNumericalSoftware`: The default numerical software,
- `./src/Server`: The server,
  - `./src/Server/Condor`: The Condor server,
  - `./src/Server/Standard`: The standard server,
  - `./src/Server/ScaLAPACK`: The ScaLAPACK server,
  - `./src/Server/PETSC`: The PETSC server.
- `./src/Testing`: The testing programs for the C, Fortran and Matlab interfaces,
- `./src/Tool`: The command line tools.

3 Networking

In NetSolve 1.2 networking is done with TCP/IP and the socket layer. However, all the networking is kept isolated from the rest of the software. The only routines performing any networking tasks are in:

- `./src/CoreFunctions/socketutil.c`
- `./src/CoreFunctions/communicator.c`

The first file contains wrappers around the socket layer to (i) bind a socket to a port; (ii) connect a socket to a remote port; (iii) poll a socket to see if some data has arrived. The wrappers are useful because they isolate those system-dependent functionalities and because the actual calls with the socket layer are rather cumbersome. The second file contains all the functions that are used to actually transfer data over the network in NetSolve. The following two sections give details on how transfers are performed.

3.1 XDR

The common method used to transfer data between machines which do not have the same internal data representations is the XDR protocol [9]. Since NetSolve operates in heterogeneous environment it uses XDR. However, XDR might be expensive when transferring large amount of data, typically user data. NetSolve is designed such that it avoids using XDR when it would be too costly and unnecessary. Each host in the NetSolve system is described by the `HostDesc` data structure (see Section 4), which contains an integer field, the `data_format`. This integer is set in the same way it is set in the reference implementation of PVM [10] in `./src/include/netsolvearch.h`. NetSolve compares the data format of hosts to decide on whether XDR should be used or not.

We give here a few notes about the use of XDR in NetSolve. First, `xdr_vector()` is used as opposed to `xdr_array()`. Indeed, `xdr_array()` inserts an XDR-encoded integer representing the size of the array before the XDR-encoded elements of the array. This is not practical when sending over the network a matrix with a number of rows different from its leading dimension (sub-matrices). Second, `xdrstdio_create()` is not used. It would be convenient in order to bind the XDR stream to the socket stream, however, this routine is not available on all platform and especially on Windows systems. Instead, `xdrmem_create()` is used with dynamically allocated buffers. It would be possible to use one static buffer for better performance. Third, `./src/CoreFunction/communicator.c` contains a function called `setXDRSizes()` which computes the memory space needed to encode each data type. The memory space needed can then be then subsequently accessed by a call to `netsolve_xdrsizeof()`. This is used to allocate the buffers in which encoded data will be placed. Again, it would be better to use `xdr_sizeof()` but it is missing in some implementations of XDR (e.g. HP-UX). Lastly, NetSolve defines the structures `scomplex` and `dcomplex` in `./include/communicator.h` to stored single and double precision numbers in a Fortran manner. The routines to process those structures with XDR are `xdr_scomplex()` and `xdr_dcomplex()` and they are implemented in `./src/CoreFunctions/communicator.c`.

3.2 Transactions

In this section, we describe a typical *transaction* between two processes over the network. By transaction we mean an entire exchange of data between the two processes, starting from socket connection until socket shutdown, with any number of data transmissions in any direction in between. Let us call A the *client* process connecting to B, the *server* process. First, B needs to set up a listening socket bound to a port with a call to `establishSocket()` and accept connections with the `accept()` system call. Then, A calls `connectToSocket()` to connect to the listening socket of B. At this point, the two processes are connected and can start calling the routines in `./src/CoreFunctions/communicator.c`. Process A calls `initTransaction()` and B calls `acceptTransaction()`. These calls take care of the agreement about the XDR encoding by sending and receiving a byte with two possible bit patterns: (i) all 0 meaning non-XDR and (ii) all 1 meaning XDR. Each call returns a `Communicator` structure on each side. That structure needs to be used for any subsequent communication until socket shutdown. At this point, both processes can exchange data by any calls to routines such as `sendInt()`, `recvInt()`, `sendArray()`, `recvArray()`, and the like, which are all implemented in `./src/CoreFunctions/communicator.c`. When all the necessary data has been transmitted, the connection must be shutdown by a call to `endTransaction()` on both sides.

3.3 Future of Networking in NetSolve

As seen in the previous sections, the networking subsystem in NetSolve is isolated from the rest of the software as it is entirely implemented in only two source files. We anticipate that this implementation directly on top of TCP/IP will become obsolete as soon as an appropriate communication protocol becomes available on the Grid. Such a protocol will probably implement secure network communications [11]. At the time this document is being written, Nexus [12] seems to be the most likely candidate as it is part of a major Grid infrastructure project [13] and already implements mechanisms for security and remote process creation in a portable fashion.

4 Fundamental Data Structures

In this section, we give brief descriptions of some of the fundamental data structures used throughout the NetSolve code. Those data structures are defined in the header files located in `./include`:

- `AgentDesc`: contains information about an agent. At the moment, it contains only a port number and a pointer to a `HostDesc`.
- `ServerDesc`: contains information about a server. That information includes a pointer to a `HostDesc`, a port number, statistics about network speed and CPU load, along with other data gathered from the server configuration file.
- `HostDesc`: contains information about a host, including its hostname, its IP address, its architecture type, ...
- `ProblemDesc`: describes a NetSolve problem and contains the problem name, description, a list of input `Object` structures, a list of output `Object` structures, along with miscellaneous information that corresponds to the content of the associated problem description file.
- `Object`: describes a datum. It contains the object type, the object's data type, a description, a name, and attributes that depend on the object type. The attributes can be filled in with information about the memory space to transfer data over the network, or left empty in which case the `Object` structure provides only problem specification information.
- `MappingDesc`: we call *mapping* the correspondence between a server and a problem. The NetSolve agent keeps track of which server can perform which problem in a matrix of mappings. Hence, a mapping contains a pointer to a `ProblemDesc`, a pointer to a `ServerDesc`, a the number of failures encountered for that particular server/problem combination.

A collection of functions is implemented in `./src/CoreFunctions` to manipulate those structures. For example, there are functions to allocate/free memory for each of the data structures, to send/receive the structures over the network, to read/write the data structures to files, etc. It would be too tedious to give here an exhaustive list of all the structures and associated functions, and we encourage the reader to just inspect the content of `./include` and `./src/CoreFunctions`.

5 Protocols

During a transaction between two processes, NetSolve uses integer *tags* for control informations (as opposed to actual data). All the tags are defined as macros in `./include/protocol.h` and start with `NS_PROT_` in order to differentiate them from other integer macros in the source code. This section is rather long as it contains the complete specification of the NetSolve protocol of the current NetSolve version. We assume that the reader is familiar with the roles of the NetSolve client, agent, and server. Each transaction is described separately, and we assume that the network connection is established and that the transaction has been initiated as explained in Section 3.2.

In what follows, we describe a transaction by specifying the sender of each datum, the data type (C data type or NetSolve-defined structure) and a short description of the datum. We use the symbol $*$ to denote zero or more datum of a current data type.

We distinguish two classes of transactions: (i) the ones that do not involve any client process and (ii) the ones that involve client processes. Readers only interested in building a new client interface to NetSolve should skip Section 5.1 and go directly to Section 5.2. All the tables referenced are in Appendix A.

5.1 Transaction not Involving any Client Process

Server registration : A new server registers to an agent according to Table 1. The server may then register to some of the agents whose descriptors are in the returned list. This is decided by the server configuration file ($*$ after the `@AGENT` clause). Note that the descriptor of the agent that was contacted in the first place is also in that list.

New agent : A new agent may let an existing server know of its existence according to Table 2. That agent was able to learn of the server's existence from another agent because the server was configured to allow such behavior ($*$ after the `@AGENT` clause in the server configuration file).

Network measurements : A process can measure the network latency and bandwidth between itself and a server according to Table 3. This feature will most certainly be rendered obsolete by the use of Grid-specific tool to obtain such measurements (see [14] for example). Table 3 does not describe the entire protocol for the actual measurements but points to the source code.

Network measurement report : A process that has completed a network measurement (see above) may report the measurement to an agent according to Table 5.

Server Re-registration : It is possible for an agent to be contacted by a server (for a workload report typically) that it is not aware of. For instance, this happens when an agent is restarted and servers never stopped running. In this case, the agent asks unknown servers to register again according to Table 4.

Terminate Server : It is possible to terminate a server according to Table 6. The process trying to terminate a server must proceed via an agent. This is used by the NetSolve command line tools for instance.

Service completion : When a server's child process finishes a user computation (successfully or not), it notifies the server according to Table 7.

Agent registration : A new agent (Agent 2) registers to an existing agent (Agent 1) according to table 8. Agent 1 send the server descriptors of those servers that were configured with a $*$ after the `@AGENT` clause of their configuration file. Agent 2 may then notify those servers of its existence.

Workload report : A server may report its workload to any agent according to Table 9.

Terminate Agent : It is possible to terminate an agent according to Table 10. This is used by the NetSolve command line tools for instance.

5.2 Transactions Involving a Client Process

Number of Servers : A process (typically a client) can query an agent to know the number of servers that (i) can solve a given problem and (ii) have never failed while solving that problem before. This is done according to Table 11

Problem Information : A process (typically a client) can query an agent to get the entire `ProblemDesc` structure associated with a problem name according to Table 12.

List of Agents : a process can query an agent to get the list of all agents in the system according to Table 13.

List of Server : a process can query an agent to get the list of all servers in the system according to Table 14.

List of Problems : a process can query an agent to get the list of all problems solvable by the system according to Table 15.

Submitting a request to an agent : a client submits a request to an agent according to Table 16. The agent does not send back the whole `ServerDesc` structures since it could contain extensive workload and network history information in future implementations of NetSolve.

Reporting a server failure : a process may report a server failure to an agent according to table 17.

Reporting a request completion : a client process must report request completions to its agent according to Table 18.

Submitting a request to a server : a client may submit a request to a server according to Table 19.

Terminating a request : a client may prematurely terminate a pending request by contacting the server serving the request according to Table 20.

6 The NetSolve Client

6.1 Client Core

As depicted on figure 1, all but the Java client interfaces are build on top of a common set of routines called the *Client Core*. Those routines are located in `./src/ClientCore` and basically implement the following functionalities:

1. sending a request to NetSolve,
2. waiting for a request's completion,
3. polling for a request,
4. getting miscellaneous information about the NetSolve system,
5. reporting errors to a NetSolve agent.

The client core routines use the data structures described in Section 4 and the networking subsystem to implement the protocol of Section 5. They also use an additional data structure, `RequestDesc`, that contains information about pending requests. That structure is defined in `./include/requestdesc.h` and is used only inside the client core. Finally, let us note that that function `netsolveWaitProbeRequest()` that is used to check on pending requests performs automatic resubmission of requests in case of failures and may call itself recursively. The purpose of this behavior is to isolate failure detection and recovery inside the client core. The role of the AIPs described in the following sections is to gather information from the end-user about the data layout in his/her memory space, process and pass down that information down to the client core.

Finally let us note that in the current implementation, a NetSolve client maintains a TCP/IP connection to each server that is performing a computation on behalf of that client. This solution was adopted for the sake of simplicity. However, it is not scalable as most operating systems impose an upper bound on the number of file descriptors that can be opened by a single process. This is especially penalizing for NetSolve's request farming feature (see Section 6.6). The alternative is to allow the client to set up a listening socket bound to a given port and have the server connect back to that socket when they complete a computation. At the moment, NetSolve implements a function, `getMaxNumberFileDesc()`, to find out how many file descriptors can be opened simultaneously by a single process. The system call `getdtablesize()` is not used because it is not quite portable.

6.2 C/Fortran API Implementation

The C and fortran APIs are implemented in `./src/CFortran`. The Fortran API consists of C functions that are to be called directly from Fortran. The main functions, `netsl()` and `netslnb()` for C, `fnetsl()` and `fnetslnb()` for Fortran, take a variable number of arguments. The differences between the C and Fortran functions come from the differences between stacks generated by C calls and Fortran calls (call by reference in Fortran, and call by value in C). In order to re-use code as much as possible, functions to transform a C or Fortran stack into an array of pointers or integers are implemented in `./src/CFortran/callingsequence.c`. Once the conversion has taken place, it is possible for all four main functions to call `netslX()` of `netslnbX()` that are independent on the original language. Those two last functions use the client core routines to make transactions with the agent and the servers.

Finally, let us note that the way a Fortran stack is build is machine dependent when arguments contains strings. Since Fortran strings are not null-terminated, it is necessary to put the length of each string on the stack. On most architecture with most compilers the lengths of all the strings passed as arguments are put at the end of the stack. However, on CRAY machines, the length of a string is put on the stack right after the string pointer. In all the cases we have encountered so far, the string lengths are put on the stack a integers and not as addresses of integers. Such details are handled by the functions in `./src/CFortran/callingsequence.c`.

6.3 Matlab API Implementation

The Matlab API to NetSolve is implemented in `./src/Matlab`. It consists of 4 mex-files, each of them implementing one function of the API. We use the mex routines provides by Matlab to access data from the Matlab space and pass them down to the client core functions. Since

Matlab is object oriented, the implementation is not as involved as for the C and Fortran APIs. However, care must be exerted when manipulating dynamic memory in Matlab. Indeed, dynamic memory must be allocated with a call to `mxCalloc()` instead of `calloc()`. Furthermore, if memory needs to be persistent between calls to the mex-files, it must be made persistent explicitly with calls to `mexMakeMemoryPersistent()`. Persistent memory is the only way to allow the user to get control back when he/she uses non-blocking calls. To that end, the Matlab API contains functions to make some of the NetSolve structures persistent.

Matlab stores complex matrices in a different way than Fortran. A complex matrix in Matlab consists of two matrices: real part and imaginary part. This means that the real part and imaginary part of a matrix element are not contiguous in memory. NetSolve assumes a Fortran storage so that it can use directly most numerical software on the server side. The Matlab API performs translation from one storage mode to the other. Finally, note that all numerical data in Matlab is stored as double precision reals. For instance, a matrix of integers is stored as a matrix of doubles. The Matlab API performs data conversions to handle this particularity.

6.4 Mathematica API Implementation

The Mathematica [15] API in implemented in `./src/Mathematica` and is very similar in philosophy to the Matlab interface. It was developed by Alexander Karaivanov and details on the implementation can be found in [5]. Let us just say that the C code from the client core function can be used more directly than for the Matlab API as the memory management issues are much more straightforward.

6.5 Java GUI/API Implementation

The Java interfaces to NetSolve are implemented 100% in Java which makes them more difficult to maintain as they cannot re-use any code from the client core. Most functions from `./src/CoreFunctions` and `./src/ClientCore` have been re-implemented in Java and are used by both the API and the GUI. At the time this document is being written, the Java interfaces have not yet been converted to NetSolve 1.2.

6.6 Farming Implementation

The `netsl_farm()` function initiates multiple NetSolve requests and takes a variable number of arguments. Like the functions of the C API, it converts its call stack to an array of pointers and integers and calls `netsl_farmX()`. Even though this is not motivated by the existence of a Fortran interface, it is always more convenient to work with an array than with a call stack. As seen in the Users' Guide, the arguments to `netsl_farm()` are values returned by calls to `ns_int()`, `ns_int_array()`, or `ns_ptr_array()`. Those functions all return an `Iterator` structure. That structure encapsulates information about how to generate the values of the arguments for each individual NetSolve request.

The scheduling strategy for farming is entirely implemented in function `netsl_farmX()` and is isolated from NetSolve's internals. This allows to do experiments and research on scheduling without having to know any of the NetSolve specifics. Furthermore, any of that research is applicable to any other system that bears fundamental similarities to NetSolve (e.g. Ninf [16]).

7 The NetSolve Agent

The NetSolve agent is implemented in `./src/Agent` as a daemon that maintains a database of which computational services are available, on which machines. In addition is keeps track of the status of the machines in terms of network proximity and workload. That data base is stored as a matrix of `MappingDesc` structures (see Section 4). The tasks performed by the NetSolve agent are of the three following types:

- Update the database with new information,
- Answer queries about the database,
- Use the database to estimate execution times.

These tasks are accomplished according to the protocol described in Section 5. Updating the database is done when (i) a new server registers, (ii) a client reports a failure about a server, (iii) a process reports a network speed measurement, (iv) a server broadcasts its workload. Queries about the database are issued by clients.

The estimation of server execution times occurs for each incoming client request and is done in `./src/Agent/scheduler.c`. For each incoming request, the agent computes an estimate of the time necessary to ship the input data, perform the computation, and retrieve the output data, for each server in the system. This estimation exploits the database (workload, network speed, computational complexity) and the user request (data size, problem size). Once an execution time has been estimated for each server, the servers can be ranked from the most suitable one to the least suitable one. That list is then returned to the client as shown in Table 16.

The implementation of the NetSolve agent is rather straightforward as it is not a real scheduler, but more a monitor of the resource pool. Future versions of NetSolve will need more sophisticated scheduling policies as the diversity of computational resources and applications increases. Such evolutions might increase the complexity of the agent (see Section 9).

8 The NetSolve Server

Like the agent, the NetSolve server is implemented as a daemon. However, it's design is a little more complex due to the fact that (i) the server monitor the workload of the host it is running on and that (ii) it can start computational processes to answer users' requests. Monitoring the workload is done by a process implemented in `./src/Server/workload_manager.c`.

This process wakes up every IDLE_TIME seconds (defined in ./include/workloadmanager.h, assesses the current workload, and may decides to broadcast its value to the agents in case of significant changes. This process needs to be restarted each time a new agent appears in the system, which is of course done automatically by the server. Starting computational processes is a little more complex and is the object of the following section.

8.1 Customized Servers

When a server finally agrees to perform a computation (after checking the workload threshold, the user access restrictions, etc..), it needs to start a *child* process. In the most common scenario, this is done by calling `fork()` and `exec()` system calls. However, several customized version of the NetSolve server use different mechanisms to spawn computing processes. At the moment, there are four versions of the NetSolve server implemented in the following directories:

1. `./src/Server/Standard`: standard `fork()` and `exec()`,
2. `./src/Server/Condor`: Condor job with `condor_submit`,
3. `./src/Server/ScaLAPACK`: MPI job with `mpirun`,
4. `./src/Server/PETSC`: MPI job with `mpirun`.

Adding a new customized server is rather standard. The procedure consists in creating a new sub-directory in `./src/Server` that implements (i) the function that spawns the computation process, (ii) the main function of the computation process. For instance in the case of the Condor [17, 18, 19] server, the spawning is done by issuing a call to the `condor_submit` executable and waiting for that call to complete, whereas the main computational function has to read its input from files rather than from the network. Once those two functions have been created, it just suffices to modify the file `./src/Server/generateservice.c` to add the call to spawn new customized server.

8.2 Expanding a Server

As explained in the NetSolve's Users' Guide, it is possible to expand a NetSolve server by generating new description files and compiling them into a new server with the code generator. The code generator is implemented in `./src/CodeGenerator`. It parse the server configuration file to get the list of description files to be used. It then parses each problem description in those files performing error checking and code generation. For each description file `./problems/file`, the code generator generates `./src/Server/numerical-file.c` along with the makefiles to compile it (see Figure 2). The generation merely replaces occurrences of mnemonics in the pseudo-code section of the problem description file by actual data structures references that are meaningful to the NetSolve server. Again, we expect that the reader is entirely familiar with the Users' Guide and we do not give details about the problem description files. At the time this document is being written, the Java applet to generate description files in an interactive manner is still under testing.

A large part of the NetSolve source code, both on the client and the server side and of course within the code generator itself, is dedicated to the parsing/interpretation/storage of the information contained in the problem description files. That part of the source code is also the most involved as the description language is complex, because low-level. Distributing the aforementioned Java applet is an attempt to provide a higher level tool to generate description file. Other projects like Ninf [16] use much higher level description languages thereby choosing convenience over generality. Indeed, it is our experience that most legacy numerical code cannot be described accurately enough by a language that does not provide low-level primitives to access the memory layout. At this time, a collaboration the the Ninf team has been initiated in order to make concerted decisions about the description language that should be used. That collaboration will undoubtedly result in changes that will impact the NetSolve code generator, server, and client.

8.3 User Provided Functions

The User Provided Function (UPF) feature in NetSolve allows a server to compile C or Fortran source code on the fly, perform some basic security checks, and link it into the computational process. This is useful for the numerous numerical software routines that take a function pointer as argument (typically for non-linear computations). The implementation of the UPF handling at the server side is done in `./src/Server/netsolveupf.c`. That files defines a list of *allowed* system or library calls. Once the source code is downloaded on the server side, the server generates a makefile to compile it. The server then examines the compiled object files (with `nm`) for undefined symbols that are not in the list of allowed calls. If all the calls are allowed, then the server generates another makefile to re-link the computational process with the UPF object files. Once that new executable is available, it can then be started by the server is the usual way (see Section 8.1). We do not make the claim that this procedure is *safe*, but it provides a basis for experimentations.

9 Conclusion

NetSolve 1.2 is a consequent improvement over version 1.1.b as the code is easier to maintain and upgrade. A number of new features have been added without any difficulty thanks to the new software architecture. As mentioned throughout this document, many parts of the software will probably be replaced in the near future (build procedure, networking, description language, etc...), but we are confident that such replacements will be rather straightforward. On issue that seems to be emerging is that of scheduling. At the moment, the NetSolve agent just maintains a database about the computational resources and uses that database to provide the client with estimation about relative execution times. The client is the one responsible for the scheduling, especially in the farming interface. Should the agent be the center of the scheduling decision ? The answer to such questions will probably arise from future experiments with the system and with new applications. Another important issue concerns *data locality and proximity*. Indeed, a number of applications do not need to use a full RPC paradigm as intermediary data might not be needed between calls. The idea would then be to *cache* such data on storage servers rather than returning it to clients. It may also be the case that an application makes a large number of calls to NetSolve and some input data is passed to each call. Such data could then be shared between multiple remote servers without having each of them download and store a copy. This is particularly easy to do if that input data is a file. This exact situation is in fact common among many embarrassingly parallel applications that would make use of farming (e.g. Monte-Carlo simulations in MCell). NetSolve's characteristics make it an ideal terrain for such computer science research as well as a powerful enabling technology that targets domain scientists.

References

[1] Ian Foster and Carl Kesselman, editors. *The Grid, Blueprint for a New computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1998.

[2] H Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 1997.

[3] H Casanova and J. Dongarra. The use of Java in the NetSolve project. In *Proc. of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics, Berlin*. Department of Computer Science, University of Tennessee, Knoxville, 1997.

[4] Henri Casanova and Jack Dongarra. NetSolve's Network Enabled Server: Examples and Applications. *IEEE Computational Science & Engineering*, 5(3):57–67, September 1998.

[5] H. Casanova, J. Dongarra, A. Karaivanov, and J. Wasniewski. Mathematica Interface to NetSolve. Technical Report UNIC-98-05, UNI-C, September 1998.

[6] H. Casanova, J. Dongarra, and K. Seymour. Client User's Guide to Netsolve. Technical Report CS-96-343, Department of Computer Science, University of Tennessee, 1996.

[7] A. Oram and S. Talbott. *Managing Projects with* `make`. O'Reilly & Associates, Inc., 1991.

[8] H. Casanova, M. Kim, J. Plank, and J. Dongarra. Request Farming with NetSolve. Technical Report to appear, Department of Computer Science, University of Tennessee, 1998.

[9] Inc. Sun Microsystems. XDR: External Data Representation Standard. RFC 1014, Sun Microsystems, Inc., Jun. 1987.

[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM : Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press Cambridge, Massachusetts, 1994.

[11] A. Freier, P. Karlton, and P. Kocher. The SSL Protocol. Technical report, Netscape Communications, 1996. Internet Draft.

[12] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[13] I. Foster and K Kesselman. Globus: A Metacomputing Infrastructure Toolkit. In *Proc. Workshop on Environments and Tools*. SIAM, to appear.

[14] R. Wolski. Dynamically forecasting network performance using the network weather service. Technical Report TR-CS96-494, U.C. San Diego, October 1996.

[15] S. Wolfram. *The Mathematica Book, Third Edition*. Wolfram Median, Inc. and Cambridge University Press, 1996.

[16] S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka, and U. Nagashima. Ninf : Network based Information Library for Globally High Performance Computing. In *Proc. of Parallel Object-Oriented Methods and Applications (POOMA), Santa Fe*, 1996.

[17] M. Litzkow, M. Livny, and M.W. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. of the 8th International Conference of Distributed Computing Systems*, pages 104–111. Department of Computer Science, University of Winsconsin, Madison, June 1988.

[18] M. Litzkow and M. Livny. Experience with the Condor Distributed Batch System. In *Proc. of IEEE Workshop on Experimental Distributed Systems*. Department of Computer Science, University of Winsconsin, Madison, 1990.

[19] J. Pruyne and M. Livny. A Worldwide Flock of Condors : Load Sharing among Workstation Clusters . *Journal on Future Generations of Computer Systems*, 12, 1996.

A Protocol specifications

Sender	Data	Content
Server	`int`	`NS_PROT_SV_REGISTER`
-	`ServerDesc`	this server's descriptor
-	`int`	number of problems for this server
-	`ProblemDesc*`	corresponding problem descriptor list
Agent	`int`	`NS_PROT_REGISTRATION_REFUSED` (abort) or
		`NS_PROT_REGISTRATION_ACCEPTED` (continue)
-	`int`	total number of known agents
-	`AgentDesc*`	corresponding agent descriptor list

Table 1: Server registration

Sender	Data	Content
Agent	`int`	`NS_PROT_NEW_AGENT`
-	`AgentDesc`	this agent's descriptor

Table 2: New agent

Sender	Data	Content
Process	`int`	`NS_PROT_PONG_REQUEST`
-	`ProblemDesc*`	corresponding problem descriptor list
Process/Server	`char*`	(see `./src/CoreFunctions/pong.c`
Server	`int`	latency in microseconds
-	`int`	bandwidth in byte per second
-	`int`	date (in seconds since 1/1/1970, 00:00:00)

Table 3: Network measurements

Sender	Data	Content
Agent	int	NS_PROT_REGISTER_AGAIN

Table 4: Network measurements

Sender	Data	Content
Process	int	NS_PROT_NETWORK_REPORT
-	IPaddr_type	IP address of the measuring process
-	IPaddr_type	IP address of the server
-	int	latency in microseconds
-	int	bandwidth in byte per second
-	int	date (in seconds since 1/1/1970, 00:00:00)

Table 5: Network measurement report

Sender	Data	Content
Process to Agent	int	NS_PROT_KILL_SERVER
-	IPaddr_type	IP address of the server to terminate
Agent to Process	int	NS_PROT_SERVER_PORT
	int	port number of the server to kill (-1 if error)
Process to Server	int	NS_PROT_KILL_SERVER
Server to Process	int	NS_PROT_NOT_ALLOWED (not allowed) or NS_PROT_KILLED (killed)

Table 6: Network measurements

Sender	Data	Content
Process	int	NS_PROT_SERVICE_FINISHED
-	int	restriction index
Server	int	any integer, for ack

Table 7: Service completion

Sender	Data	Content
Agent 2	int	NS_PROT_AG_REGISTER
-	AgentDesc	this agent's descriptor
Agent 1	int	NS_PROT_REGISTRATION_REFUSED (abort) or
		NS_PROT_REGISTRATION_ACCEPTED (continue)
-	int	total number of known agents
-	AgentDesc*	corresponding agent descriptor list
-	int	total number of known and advertisable servers
-	ServerDesc*	corresponding server descriptor list
-	int	total number of known problems
-	ProblemDesc*	corresponding problem descriptor list
-	int	total number of known mappings
-	MappingDesc*	corresponding mapping descriptor list

Table 8: Agent registration

Sender	Data	Content
Server	int	NS_PROT_WPRKLOAD_RREPORT
-	IPaddr_type	the server's IP address
-	int	the server's port number
-	int	the server's workload
-	int	the date (in seconds since 1/1/1970, 00:00:00)

Table 9: Workload report

Sender	Data	Content
Process	int	NS_PROT_KILL_AGENT
-	char	username
Server	int	NS_PROT_NOT_ALLOWED (unallowed)
	int	NS_PROT_KILLED (killed)

Table 10: Terminate Agent

Sender	Data	Content
Process	int	NS_PROT_NB_SERVERS
-	char	problem's name
Agent	int	number of servers (may be 0)

Table 11: Number of Servers

Sender	Data	Content
Process	`int`	`NS_PROT_PROBLEM_INFO`
-	`char`	problem's name
Agent	`int`	`NS_PROT_PROBLEM_NOT_FOUND` (abort) or
		`NS_PROT_PROBLEM_PROBLEM_DESC` (ok)
	`ProblemDesc`	number of servers (may be 0)

Table 12: Problem Information

Sender	Data	Content
Process	`int`	`NS_PROT_AGENT_LIST`
Agent	`int`	number of agents
-	`AgentDesc*`	corresponding agent descriptor list

Table 13: List of Agents

Sender	Data	Content
Process	`int`	`NS_PROT_SERVER_LIST`
Agent	`int`	number of servers
-	`ServerDesc*`	corresponding server descriptor list

Table 14: List of Servers

Sender	Data	Content
Process	`int`	`NS_PROT_PROBLEM_LIST`
Agent	`int`	number of problems
-	`ProblemDesc*`	corresponding problem descriptor list

Table 15: List of Problems

Sender	Data	Content
Process	int	NS_PROT_PROBLEM_SUBMIT
-	char *	problem name
-	int	input size in bytes
-	int	output size in bytes
-	int	problem size
Agent	int	NS_PROT_PROBLEM_NOT_FOUND (abort) or NS_PROT_OK
-	int	number of servers
-	(char * +	server hostname
	IPaddr_type +	server IPAddr
	int +	server port number
	int +	server data format
	int)*	predicted execution time in seconds

Table 16: Submitting a request to an agent

Sender	Data	Content
Process	int	NS_PROT_SV_FAILURE
-	IPaddr_type	server's IP address
-	char *	server's hostname
-	int	HOST_ERROR or SERVER_ERROR

Table 17: Reporting a server failure

Sender	Data	Content
Client	`int`	`NS_PROT_JOB_COMPLETED`
-	`IPaddr_type`	successful server's IP address

Table 18: Reporting a request completion

Sender	Data	Content
Client	`int`	`NS_PROT_PROBLEM_SOLVE`
-	`ProblemDesc`	descriptor of the problem to solve
Server	`int`	`NS_PROT_PROBLEM_NOT_FOUND` (abort) or `NS_PROT_BAD_SPECIFICATION` (abort) or `NS_PROT_NOT_ALLOWED` (abort) or `NS_PROT_ACCEPTED` (ok)
Client	`int`	client major
-	`char *`	agent name
-	`Object*`	all the input objects (with data)
Server	`int`	`NS_PROT_SERVIVE_PID` (ok) or `NS_PROT_INTERNAL_FAILURE` (abort) or `NS_PROT_BAD_VALUES` (abort) or `NS_PROT_DIM_MISMATCH` (abort) or `NS_PROT_NO_SOLUTION` (abort) or `NS_PROT_UPF_ERROR` (abort) or `NS_PROT_UPF_UNSAFE` (abort)
-	`int`	service process pid
Computation under way		
Server	`char *`	server `stdout`
-	`int`	`NS_PROT_SOLVED` (ok) or `NS_PROT_INTERNAL_FAILURE` (abort) or `NS_PROT_BAD_VALUES` (abort) or `NS_PROT_DIM_MISMATCH` (abort) or `NS_PROT_NO_SOLUTION` (abort)
-	`Object*`	all the output objects (with data)

Table 19: Submitting a request to a server

Sender	Data	Content
Client	`int`	`NS_PROT_KILL_REQUEST`
-	`int`	service process pid
Server	`int`	`NS_PROT_KILLED` (ok) or `NS_PROT_SV_FAILURE` (error)

Table 20: Terminating a request