NetSolve: A Network Server for Solving Computational Science Problems

Henri Casanova[*] Jack Dongarra[*†]

April 26, 1996

Abstract

This paper presents a new system, called NetSolve, that allows users to access computational resources, such as hardware and software, distributed across the network. The development of NetSolve was motivated by the need for an easy-to-use, efficient mechanism for using computational resources remotely. Ease of use is obtained as a result of different interfaces, some of which require no programming effort from the user. Good performance is ensured by a load-balancing policy that enables NetSolve to use the computational resources available as efficiently as possible. NetSolve offers the ability to look for computational resources on a network, choose the best one available, solve a problem (with retry for fault-tolerance), and return the answer to the user.

Keywords

Networking, Heterogeneity, Load Balancing,
Client-Server, Fault Tolerance, Numerical Computing, Virtual Library.

---

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

1 Introduction

An ongoing thread of research in scientific computing is the efficient solution of large problems. Various mechanisms have been developed to perform computations across diverse platforms. The most common mechanism involves software libraries. Unfortunately, the use of such libraries presents several difficulties. Some software libraries are highly optimized for only certain platforms but do not provide a convenient interface to other computer systems. Other libraries demand considerable programming effort from the user, who may not have the time to learn the required programming techniques. While a limited number of tools have been developed to alleviate these difficulties, such tools themselves are usually available only on a limited number of computer systems. MATLAB [1] is an example of such a tool.

These considerations motivated the establishment of the NetSolve project. NetSolve is a client-server application designed to solve computational science problems over a network. A number of different interfaces have been developed to the NetSolve software so that users of C, Fortran, MATLAB, or the Web can easily use the NetSolve system. The underlying computational software can be any scientific package, thereby ensuring good performance results. Moreover, NetSolve uses a load-balancing strategy to improve the use of the computational resources available.

This paper introduces the NetSolve system, its architecture and the concepts on which it is based. We then describe how NetSolve can be used to solve complex scientific problems. We emphasize that NetSolve can be used to solve many classes of problem.

2 The NetSolve System

This section briefly describes the NetSolve system, the protocols in use, and the issues involved in managing such a system.

2.1 Architecture

The NetSolve system is a set of loosely connected machines. By *loosely* connected, we mean that these machines can be on the same local network or on an international network. Moreover, the NetSolve system can be *heterogeneous,* which means that machines with incompatible data formats can be in the system at the same time.

The current implementation sees the system as a completely connected graph without any hierarchical structure. This initial implementation was adopted for simplicity and is definitely viable for now. Our current idea of the *NetSolve world* is of a set of independent NetSolve systems in different locations, possibly providing different services. A user can then contact the system he wishes, depending on the task he wants to have performed and on his own location. In order to manage efficiently a pool of hosts scattered on a large-scale network, future implementations might provide greater structure (e.g., a tree structure), which will limit and group large-range communications.

Figure 1 shows the global conceptual picture of the NetSolve system. In this figure, a NetSolve client send a request to the NetSolve agent. The agent "chooses" the best NetSolve resource according to the size and nature of the problem to be solved.

Several instances of the NetSolve agent can exist on the network. A good strategy is to have an instance of the agent on each local network where there are NetSolve clients. Of course, this is not mandatory; indeed, one may have only a single instance of the agent per NetSolve system.

Every host in the NetSolve system runs a NetSolve *computational* server (also called a *resource*, as shown in Figure 1). The NetSolve resources have access to scientific packages (libraries or stand-alone systems).

An important aspect of this server-based system is that each instance of the agent has its own *view* of the system. Therefore, some instances may be aware of more details than are others, depending on their locations. But eventually, the system reaches a stable state in which every instance possesses all the available information on the system (provided the system does not undergo never-ending modifications).

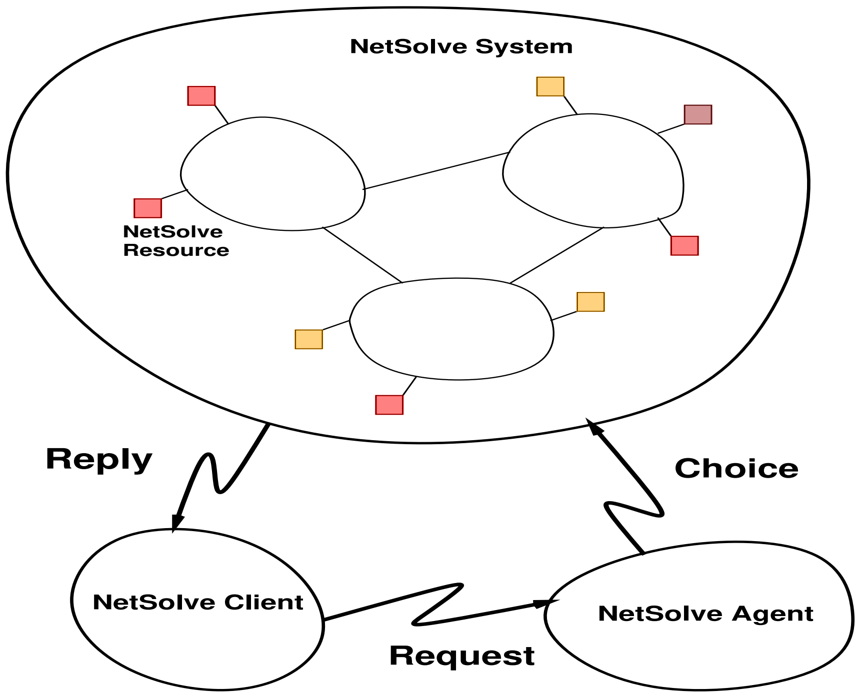

Figure 1: The NetSolve System

2.2 Protocol Choices

The communication within the NetSolve system is achieved through the socket layer. We chose to use the TCP/IP protocol because it ensures reliable communication between processes. The fact that a process is limited to a certain number of simultaneous TCP/IP connections was not a problem (given the NetSolve specification.)

To ensure the possible use of an heterogeneous environment, NetSolve uses the XDR protocol between hosts with incompatible data formats. Actually, this is the default protocol before two hosts agree that they use the same data format. Not using XDR when not necessary is actually an issue here, since we expect to transfer large amounts of data over the network (the user data).

2.3 NetSolve Management

The main philosophy behind the architecture of a NetSolve system is the following. Each NetSolve process (instance of the agent or computational resource) is an independent entity. The system therefore can be modified without endangering its integrity, since any NetSolve process can be deleted/created at any time. For instance, it is possible to have a NetSolve system with no agent. Such a system is just not accessible by any user. An instance of the agent can be restarted later on, to make the system accessible again.

Managing such a system can rapidly become difficult, and a tool is needed to have a *centralized* view of the system. To make this tool as convenient as possible, we developed two CGI scripts accessible from the World Wide Web. These scripts take as input the location of an agent instance (the name of the host where it is running) in order to identify the NetSolve system to inspect. The first script outputs the list of agent instances or computational resources available in the system (a list of host names and IP-addresses). The second script outputs the list of problems solvable within the system.

3 Computational Server Management

This section describes what a NetSolve problem is and how to configure, compile, and start a new computational resource within a NetSolve system.

3.1 Problem Specification

To keep NetSolve as general as possible, we need to find a formal way of describing a problem. Such a description must be carefully chosen, since it will affect the ability to interface NetSolve with arbitrary software.

A problem is defined as a 3-tuple: $< name, inputs, outputs >$, where

- *name* is a character string
- *inputs* is a list of input objects
- *outputs* is a list of output objects

An object is itself described as follows: $< object, data >$, where *object* can be 'MATRIX', 'VECTOR' or 'SCALAR' and *data* can be any of the standard FORTRAN data types.

This description has proved to be sufficient to interface NetSolve with numerous software packages. But it is to be remembered that NetSolve is still at an early stage of development and is likely to undergo modifications in the future.

3.2 Arbitrary Calling Sequences

We have just described what a problem is conceptually. We now need a way to describe how a problem is to be *called* by the user. Ideally, we would like users already using scientific software packages from C or Fortran to be able to switch to NetSolve with no modification to their code. From this viewpoint, when describing a problem as in the preceding subsection, we also describe what we call its *format*. This is in effect describing what the calling sequence to NetSolve for this problem should be from C or Fortran. Moreover, a problem can have different calling sequences, and the user can chose between them.

3.3 Creating a Server

Part of our design objective was to ensure that NetSolve have an extensive application range. Thus, we wished to be able to add new problems to a computational server. We considered it unacceptable to have the NetSolve administrator modify the NetSolve code itself for each new problem addition.

Two solutions were possible. We could have our server spawn off executables, or we could have it call the numerical software explicitly in its own code. The first solution seems much easier to implement: a computational server could have access to a directory containing all the executables for all the problems. This approach, however, has the following drawbacks. First, maintaining such a directory may not be easy in a distributed file system environment or in the case when some servers want to have access to only a subset of the set of problems. Second, and more important, such a design requires a stand-alone executable for each problem. Moreover, since most of the numerical softwares likely to be interfaced with NetSolve are actual libraries, it seems redundant to have our computational servers start up an executable calling the library itself. Therefore, we decided to take the second approach and have our servers directly call the underlying software.

We developed a simple tool to handle this task. The input for this tool is a configuration file describing each problem in a formal way; the output is the actual C code for the computational process in charge of the problem solving. Thus, new problems can be added without having to be concerned about the NetSolve internal data structure.

In its first version, this pseudo-compiler still requires some effort from the NetSolve administrator. In fact, since any arbitrary library is supposed to be accessible from NetSolve, we cannot completely free the administrator from code writing. We can, however, provide the administrator with a simple and efficient way of accessing the parameters to the problem. In particular, the function calls to the library have to be written in C, using a predefined set of macros.

3.4 Scientific Packages

NetSolve is able to use any scientific linear algebra package available on the platforms it is installed on, provided that the above formalism remains valid. This feature allows the NetSolve administrator not only to choose the best platform on which to install NetSolve, but also to select the best packages available on the chosen platform.

The current implementation of NetSolve at the University of Tennessee uses the BLAS [2], [3], [4], LAPACK [5], ItPack [?], and LINPACK [?]. These packages are available on a large number of platforms and are freely distributed.

The use of ScaLAPACK [6] on massively parallel processors would be a way to use the power of high-performance parallel machines via NetSolve.

4 Interfaces

One of the main goals of NetSolve is to provide the user with the largest number of interfaces and to keep them as simple as possible. We describe here the different interfaces currently available, classified into two groups.

4.1 Interactive Interfaces

Interactive interfaces offer several advantages. First, they are easy to use because they completely free the user from any code writing. Second, the user still can exploit the power of software libraries. Third, they provide good performance by capitalizing on standards tools like MATLAB. Let us assume, for instance, that MATLAB is installed on one machine on the local network. It is possible to use NetSolve via MATLAB on this machine and in fact use the computational power of another more powerful machine where MATLAB is not available.

The current implementation of NetSolve contains two interactive interfaces.

4.1.1 The MATLAB Interface

Within MATLAB, NetSolve may be used in two ways. It is possible to call NetSolve in a *blocking* or *nonblocking* fashion. Here is an example of the MATLAB interface to solve an linear system computation using the blocking call:

```
>> a = rand(100); b = rand(100,1);
>> x = netsolve('ax=b',a,b)
```

This MATLAB script first creates a random 100×100 matrix, a, and a vector b of length 100. The call to the `netsolve` function returns with the solution This call manages all the NetSolve protocol, and the computation may be executed on a remote host.

Here is the same computation performed in a nonblocking fashion:

```
>> a = rand(100); b = rand(100,1);
>> request = netsolve_nb('send','ax=b',a,b)
>> x = netsolve_nb('probe',request)
```

```
        Not Ready Yet
    >> x = netsolve_nb('wait',request)
```

Here, the first call to `netsolve_nb()` sends a request to the NetSolve agent and returns immediately with a request identifier. One can then either *probe* for the request or *wait* for it. This approach allows user parallelism and communication/computation overlapping. (see Section 7).

Other functions are provided, for example, to obtain informations on the problems available or on the status of the pending requests.

4.1.2 The Shell Interface

We also developed a shell interface. Here is the same example as above, with the shell interface:

```
earth % netsolve Ax=b A b solution
```

Here, `mat`, `real`, and `imaginary` are files. This interface is slightly different from the MATLAB interface because the call to netsolve does not make any difference between inputs and outputs. The difference is made internally, and the user must know the correct number of parameters. As mentioned before, information on the problem specifications can be obtained by using the CGI scripts (on the NetSolve Web site).

4.2 Programming Interfaces

In addition to interactive interfaces, we have developed two programming interfaces, one for Fortran and one for C. Unlike the interactive interfaces, programming interfaces require some programming effort from the user. But again, with a view to simplicity, the NetSolve libraries contain only a few routines, and their use has been made as straightforward as possible. As in MATLAB, the user can call NetSolve *asynchronously*.

Simple examples of the C and Fortran interfaces can be found in Appendix A and B.

5 Load Balancing in NetSolve

Load balancing is one of the most attractive features of the NetSolve project. Since NetSolve performs computations over a network containing a large number of machines with different characteristics, it seems logical to think that one of these machines is the most suitable for a given problem. Before we consider how NetSolve tries to determine which machine is to be chosen, let us examine what criteria determine the *best* machine.

5.1 Calculating the Best Machine

The hypothetical best machine is the one yielding the smallest execution time T for a given problem P. Therefore, we have to estimate this time on every machine M in the NetSolve system. Basically, we split the time T in T_n and T_c, where

- T_n is the time to send the data to M and receive the result over the network, and
- T_c is the time to perform the computation on M.

The time T_n can be computed by knowing the

1. network latency and bandwidth between the local host and M,
2. size of the data to send, and
3. size of the result to be received.

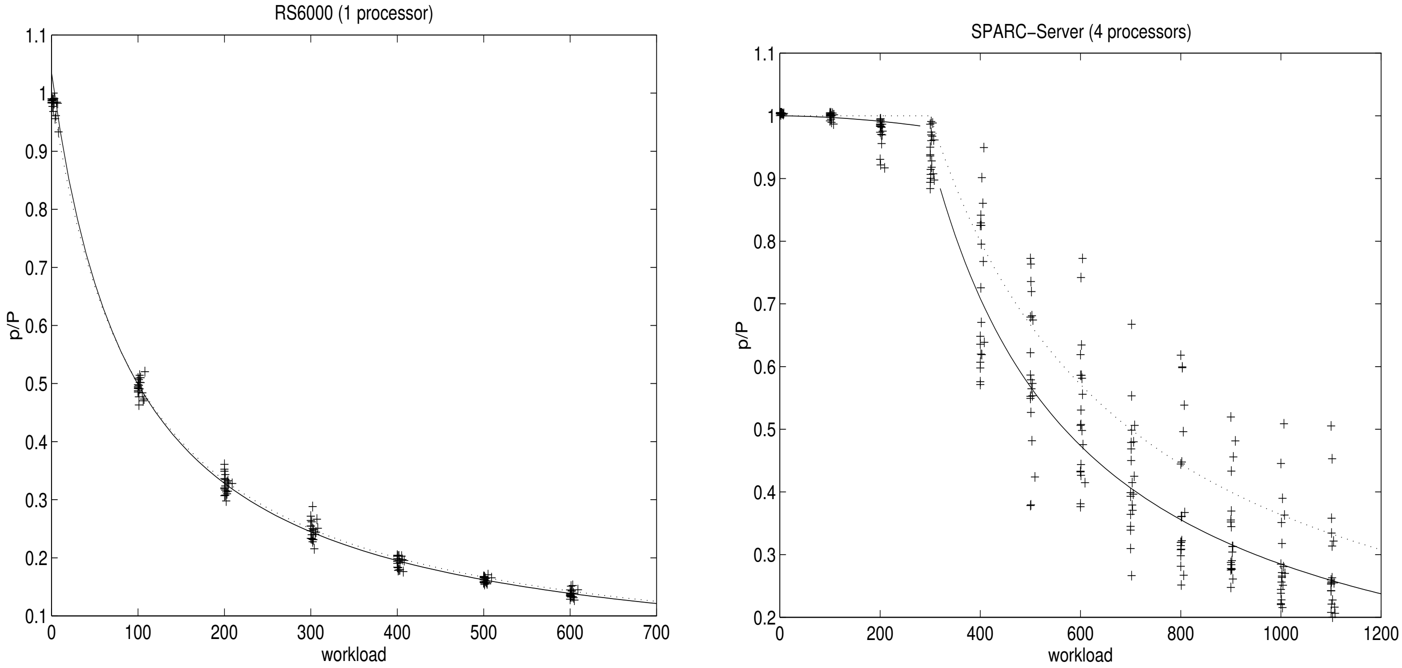

Figure 2: p/P versus workload

The computation of T_c involves knowledge of the

1. size of the problem,
2. complexity of the algorithm to be used, and,
3. performance of M, which depends on
   - the workload of M and
   - the *raw* performance of M.

5.2 Performance Model

We have developed a simple theoretical model enabling us to estimate the performance, given the raw performance and the workload. This model gives the estimated performance, p, as a function of the workload, w; the raw performance, P; and the number of processors on the machine, n:

$$p = \frac{P \times 100 \times n}{100 \times n + max(w - 100 \times (n-1), 0)}$$

To validate this model, we performed several experiments. The results of the experiments are plotted in Figure 2, which shows the ratio p/P versus the workload of the machine. Each measure gave one of the "+" marks. We then computed the mean of all the measures for every value of the workload. An asymptotic interpolation of these mean values is shown with a continuous curve. Our theoretical model is shown with the dashed line.

In Figure 2a, we can see that the theoretical model is close to the experimental results. In Figure 2b, since the machine has four processors, the beginning of the curve is a flat line, and the performance begins to drop when the four processors are loaded. Our model is less accurate and always optimistic because it does not take into account any operating system delay to manage the different processors. The widely varying behavior of the four-processor machine comes from the fact that the operating system makes process migrations between the processors.

5.3 Computation of T

The computation of T takes place on a communication server for each problem request and for each computational server M. This computation uses all the parameters listed in the preceding section. We distinguish three different classes of parameter:

- The client-dependent parameters
  1. The size of the data to send
  2. The size of the result to be received
  3. The size of the problem
- The **static** server-dependent parameters
  1. The network characteristics between the local host and M
  2. The complexity of the algorithm to be used on M
  3. The *raw* performance of M
- The **dynamic** server-dependent parameters
  1. The workload of M

The client-dependent parameters are included in the problem request sent by the client to the communication server. Their evaluation is therefore completely straightforward. The static server-dependent parameters are generally assessed once, when a new server contacts the other NetSolve servers already in the configuration.

Network Characteristics. The network characteristics are assessed several times, so that a reasonable average value for the latency and bandwidth can be obtained. We still call them *static* parameters, however, since they are not supposed to change greatly once their mean value has been computed.

Complexity of the Algorithm. When a new computational server joins the NetSolve system, it posts the complexity of all of the problems it is willing to service. This complexity does not change thereafter, since it depends only on the software used by the computational server.

Raw Performance. By *raw* performance, we mean the performance of the machine with no other process using the CPU. Its value is determined by each computational server at startup time. We use the LINPACK benchmark to obtain the Kflop/s rate. The LINPACK benchmark computes the "user time" for its run and therefore corresponds to our definition of raw performance.

5.4 The Workload Model

Workload parameters are the only dynamic server-dependent parameters required to perform the computation of the predicted execution time T.

Each instance of the agent possesses a cached value of the workload of every computational server. By *cached*, we mean that this value is directly used for T's computation and that it is updated only periodically. Admittedly, this value may be out of date and lead to an occasional wrong estimate of T. Nevertheless, we prefer on the average to take the risk of having a wrong estimate than to pay the cost for getting a constantly accurate one.

We emphasize that we have tried to make this estimate as accurate as possible, while minimizing the cost of its computation. Figure 3 shows the scheme we used to manage the workload broadcast.

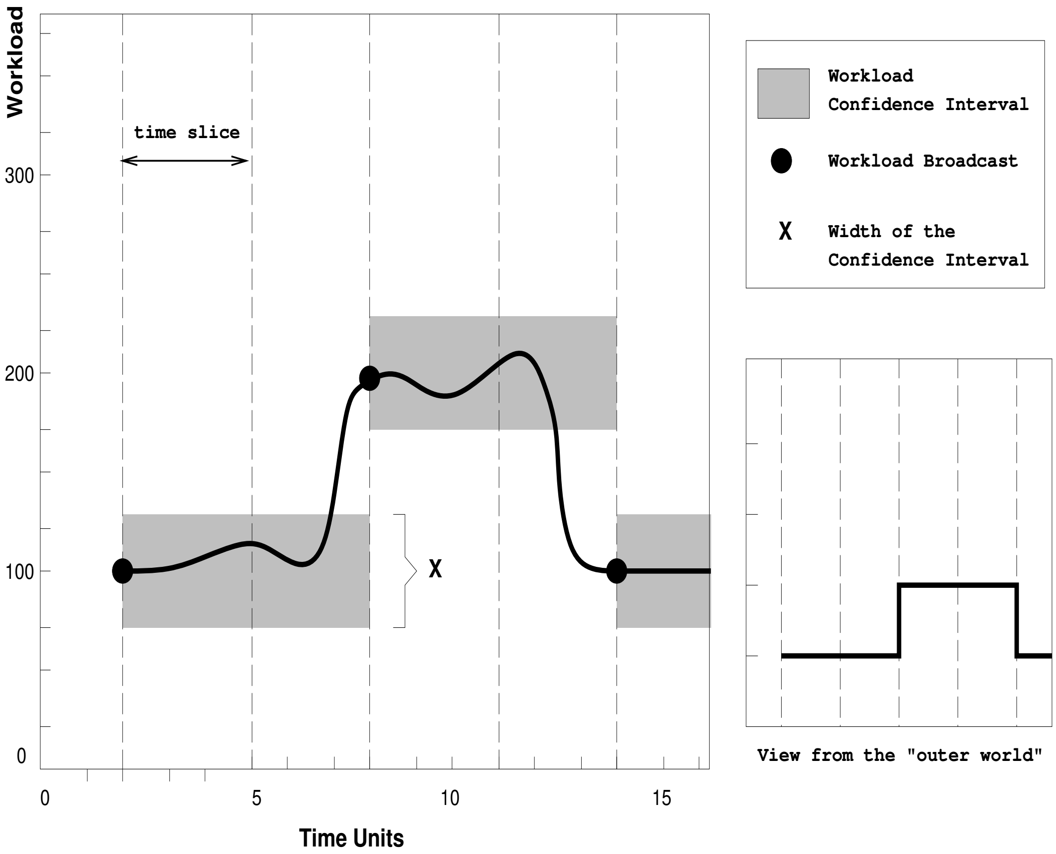

Figure 3: Workload Policy in NetSolve

Let us consider a computational server M and a communication server C. C performs the computation of T according to the last value of M's workload it knows. M broadcasts its workload periodically. In Figure 3, we call *time slice* the delay between to workload broadcast from M. This figure shows the workload function of M versus the time. The simplest solution would be to broadcast the workload at the beginning of each time slice. However, experience proves that the workload of a machine can stay the same for a very long time. Therefore, most of the time, the same value would be broadcast again and again over the network. To avoid this useless communication, we chose to broadcast the workload only when it has **significantly** changed. In the figure, we see some shaded areas called the *confidence* interval. Basically, each time the value of the workload is broadcast, the NetSolve computational server decides that the next value to be broadcast should be different enough from the last broadcast one—in other words, outside this confidence interval. In Figure 3, the workload is broadcast three times during the first five time slices.

Two parameters are involved in this workload management: the width of a time slice and the width of the confidence interval. These parameters must be chosen carefully. A time slice that is too narrow causes the workload to be assessed often, which is costly in term of CPU cycles. We have to remember that a NetSolve server is supposed to run on a host for a long period of time; it is impossible to let it monopolize a lot of CPU time. The width of the confidence level must also be considered carefully. A narrow confidence interval causes a lot of useless workload broadcasting, which is costly in term of network bandwidth.

Choosing an effective time slice and confidence interval serves another function. It helps to make the workload information on the communication servers as accurate as possible, so that the estimated value of T is reasonable. We emphasize that experimentation is needed to determine the most suitable time slice and confidence intervals.

6 Fault Tolerance

Fault tolerance is an important issue in any loosely connected distributed system like NetSolve. The failure of one or more components of the system should not cause any catastrophic failure. Moreover, the number of side effects generated by such a failure should be as low as possible and minimize the drop in performance. Fault tolerance in NetSolve takes place at different levels. Here we will justify some of our implementation choices.

6.1 Failure Detection

Failures may occur at different levels of the NetSolve protocols. Generally they are due to a network malfunction, to a server disappearance, or to a server failure. A NetSolve process (i.e., a client, a server, or a utility process created by a server) detects such a failure when trying to establish a TCP connection with a server. The connection might have failed or have reached a timeout before completion. In this case, this NetSolve process reports the error to the NetSolve agent, which takes the failure into account.

One of the prerequisites for NetSolve was that a server can be stopped and restarted safely. Therefore, all the error reports contain information to determine whether the server was restarted after the error occurred. Indeed, since NetSolve can be used over a wide area network, some *old* failure reports may very likely arrive after the server that failed has been restarted. The current policy ensures that a running server will never be seen as stopped by other servers. In other words, *a NetSolve server can always be stopped and restarted safely.*

When the agent takes a failure into account, it marks the failed server in its data structures and does not remove it. A server will be removed only after a given time, and only if it has not been restarted.

6.2 Failure Robustness

Another aspect of fault tolerance is that it should minimize the side effects of failures. To this end, we designed the client-server protocol as following. When the NetSolve agent receives a request for a problem to be solved, it sends back a list of computational servers sorted from the most to the least suitable one. The client tries all the servers in sequence until one accepts the problem. This strategy allows the client to avoid sending multiple requests to the agent for the same problem if some of the computational servers are stopped. If at the end the list no server has been able to answer, the client asks another list from the agent. Since it has reported all these failures, it will receive a different list.

Once the connection is established with a computational server, there still is no guarantee that the problem will be solved. The computational process on the remote host can die for some reason. In that case, the failure is detected by the client, and the problem is sent to another available computational server. This process is transparent to the user but, of course, lengthens the execution time. The problem is migrated between the possible computational servers until it is solved or no server remains.

6.3 Taking Failures into Account

When a failure occurs, the instances of the agent update their view of the NetSolve system. They keep track of the status of the remote hosts: reachable or unreachable. They also keep track of the status of the NetSolve servers on these hosts: running, stopped, or failed. When a host is unreachable or a NetSolve server is stopped for more than 24 hours, the agent erases the corresponding entry in their view of the NetSolve system.

The agent also keeps track of the number of failures encountered when using a computational server. Once this number reaches a limit value, the corresponding entry is removed. Therefore, if a computational server is poorly implemented, for instance because it calls a library incorrectly, it will eventually disappear from the system.

7 Performance

One of the challenges in designing NetSolve was to combine ease of use and excellence of performance. Several factors ensure good performance without increasing the amount of work required of the user. In addition to the availability of diverse scientific packages (as discussed in the preceding section), these factors include load balancing and the use of simultaneous resources.

- Load balancing. Given all the computational resources available, NetSolve provides the user with a "best effort" to find the most suitable resource for a given problem.

- Simultaneous resources. Using the nonblocking interfaces to NetSolve, the user can write a NetSolve application that has some parallelism. In Figure 4, we see the result of experiments conducted on a network of SPARC workstations. The NetSolve program kept sending requests so that ten 600×600 eigenvalues problems were solved simultaneously over the network. We also added computational servers to the NetSolve configuration while running this program. Figure 4 shows the execution time for each problem for each experiment. As expected, the problems are solved simultaneously on different servers, and the average execution time for one problem decreases when the number of computational servers increases.

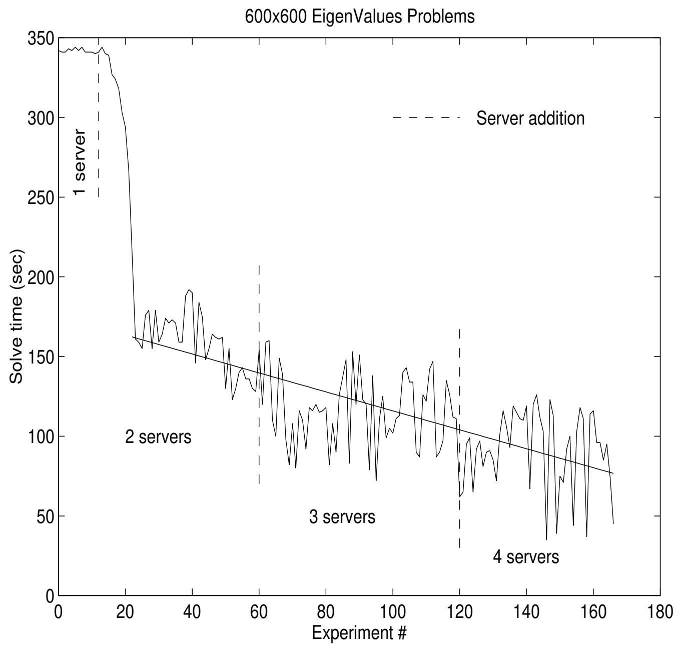

Figure 4: Simultaneous request to an evolving NetSolve system

8 Future Work

Since the NetSolve project is still at an early development stage there is room for improvement at the interface level as well as at the conceptual level.

We also have to increase the number of interactive interfaces. For instance, we could write Maple and Mathematica interfaces, similar to the MATLAB one. Currently, we are thinking of providing the user with a Java interface. Such an interface should be easy to use and immediately accessible via the Web.

The load-balancing strategy must be improved in order to change the “best guess” into a “best choice” as much as possible. The challenge is to come close to a best choice without flooding the network. The danger is to waste more time computing this best choice than the computation would have taken in the case of a best guess only. Also, we might wish to add a hierarchy in the NetSolve systems, so that a single system could cover a large-scale network efficiently.

Some new issues are raised also when trying to make NetSolve easier to interface with any arbitrary software. One of those is the “user-defined function” problem. Some scientific packages require the user to provide a function in order to solve a problem (typically with iterative methods). We are investigating different approaches to allow this in NetSolve.

All these improvements are intended to combine ease of use and performance, the main purpose of the NetSolve project.

A Example: The NetSolve C Interface

```
double A[100*100];                  /* Matrix A                                  */
double Real[100],Imaginary[100];    /* real and imaginary parts of A's eigenvalues */
int request;                        /* NetSolve request number                   */
int is_finished;                    /* Flag giving the computation status        */

/* Blocking call       */
request = netsolve("eig",           /* Eigenvalues problem             */
              A,100,                /* One matrix in input : A 100x100 */
              Real,Imaginary);      /* Two vectors in output :         */
                                    /*   Real and Imaginary            */

/* Asynchronous call   */
request = netsolve_nb("eig",A,100,
                     Real,Imaginary);

...    Some computations

is_finished = netsolve_get(request,PROBE);     /* poll the previous request       */

...    Some computations

is_finished = netsolve_get(request,WAIT);      /* poll in a blocking fashion      */
```

B Example: The NetSolve Fortran Interface

```
      INTEGER LDA,N
      PARAMETER(LDA = 100, N = 100)
      DOUBLE PRECISION A(LDA,N), R(N),I(N)
      INTEGER REQUEST,ISREADY

      * Blocking Call
      CALL FNSOLVE('eig',REQUEST,
     $             A,LDA,N,R,I)

      * Asynchronous  Call
      CALL FNSOLVE_NB('eig',REQUEST,
     $                A,LDA,N,R,I)

...    Some computations

      CALL FNSGET(REQUEST,PROBE,ISREADY)

...    Some computations

      CALL FNSGET(REQUEST,WAIT,ISREADY)
```

References

[1] Inc The Math Works. *MATLAB Reference Guide*. 1992.

[2] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 5:308–325, 1979.

[3] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–32, 1988.

[4] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[5] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM Philadelphia, Pennsylvania, 2 edition, 1995.

[6] J. Dongarra and D. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, 1995.