A Scheme for Transfer and Execution of Architecture Independent Procedures

Dheeraj Sanghi
Dept. of Computer Science & Engg.
Indian Institute of Technology Kanpur
dheeraj@iitk.ac.in

Manu Thambi
Computer Science Department
University of Washington, Seattle
manu@cs.washington.edu

Abstract

Conventional Remote Procedure Call (RPC) mechanisms allows one to pass parameters to a procedure residing in a remote machine, execute it there and obtain results. In this paper we propose a scheme which will allow one to send *the* procedure code *as well as the parameters to the remote machine and execute it. The scheme also allows a program to* import *a procedure from another machine and execute it locally. This tremendously increases the flexibility of network applications. If this scheme is implemented widely, an application will no longer be constrained to use the standard protocols to communicate to other machines which does not have a corresponding peer-application running. Whenever a standard protocol is not suitable, the application developer can design a new protocol and create a server which can be exported to the remote machine as and when required. The scheme ensures that the overheads involved are automatically reduced by caching frequently used procedures. It also provides facilities which enable an application program to search the network and obtain procedures for execution as and when required with a high degree of transparency. The scheme also deals with the security issues involved and provides mechanisms for a machine to ascertain the authenticity of an imported procedure without trusting any other machine on the network. The procedures are transfered in an architecture independent format, which allows heterogeneous machines to communicate. The format is designed to make the translation to machine code fast and the resultant executable code efficient.*

1 Introduction

Older application programs used to consider the computer network as a means of transferring data between two parts of the same application, running on different machines. A gradual shift is occurring on the application's view of the network, from a communication medium to a resource provider. The network offers all sorts of resources like file systems, online libraries, computational resources, etc. For using these resources flexibly and efficiently, it is desirable that an application be able to execute procedures on remote machines.

Presently the support available for remote execution is very limited. The conventional remote procedure call mechanisms (e.g., Sun RPC [3]) helps an application to send parameters for a procedure which resides on a remote machine, execute it there and obtain results. It is application's responsibility to make sure that the required procedure is present on the remote machine. Much more flexible network applications could be developed if it were possible for applications to send procedures to remote machines for execution. Normally an application requires that a server be running at the remote machine. This rules out applications like an FTP client using an enhanced file transfer protocol, because the user cannot make sure that the server for the enhanced protocol will be running on the remote machine. If the capability proposed in this paper is available, the client can send a server program to the remote machine if it is not present there. Therefore each application can use its own customized protocols to communicate with remote machines.

Another facility which improves flexibility of an application is the ability to request and receive procedures from machines on the network. One environment which allows this is a Java (from Sun Microsystems) enabled browser. Such a browser can request for a Java Applet from an HTTP server. The Java Applet received can be either compiled and run locally or interpreted locally. Java Applets are sent over the network as *Java Byte Codes* written for the *Java Virtual Machine* [7] (defined by Sun), which is simulated on the real machine. Thus the Java Applet code is independent of the architecture of the machine on which it is run. Java Applets are mainly intended to create animated WWW pages and hence efficiency was not the prime concern while designing the Java Virtual Machine. Moreover the capabilities of Java

Applets are severely restricted due to security concerns. It is desirable to have a scheme which could import procedures from machines on a heterogeneous network to execute locally with an efficiency comparable to that of native compiler generated code.

This paper describes a scheme which achieves the above objectives in a coherent manner. The scheme enables an application to execute a procedure (which from now on we will call a *module*) on a remote machine. It also allows an application to fetch a module from the network. Since the network is assumed to be heterogeneous, and it is impractical for every application to know the architectures of all the remote machines, the modules which are transfered between two machines should be written for a machine independent *Virtual Machine*. This Virtual Machine is simulated on a real machine. The Java Virtual Machine cannot be used because of the reasons mentioned above. Since efficiency is of prime concern, the module will be translated into the native machine code when required and executed rather than interpreted. The Virtual Machine is designed so that the translation to the real machine code of various current architectures are fast and the code generated is efficient, at the expense of making the compilation from the high level language to Virtual Machine code slower and complicated, since it is a one time affair.

The scheme includes a protocol to make requests for remote execution as well as to download modules. For this purpose the modules have to be 'named' or tagged. Since it is not practical to have a central authority who is responsible for tagging, we use the MD5 Message Digest Algorithm to form a 128 bit MD5 sum (hence forth called module ID) to be used as a tag. Using this as the tag would solve many security problems. Caching of modules can be used to improve efficiency. This will make frequently used modules run as efficiently as conventional programs.

This paper is organized as follows: Section 2 describes in detail the advantages and the flexibility the scheme offers. Section 3 gives the general outline of the scheme. Section 4 discusses the module transfer protocol in detail. Section 5 describes the Virtual Machine architecture. Section 6 discusses the Virtual Machine API in brief. Section 7 concludes.

2 Advantages

The advantages of being able to transfer code to a remote machine and execute it with low latencies are many. Here we will examine four cases where the scheme would improve performance and/or flexibility considerably. The four cases represent four different classes of applications.

- **Allows applications to use tailor made protocols to communicate with any remote machine**

  This scheme will allow applications to use new protocols specifically made for the application to communicate with any remote host, if the existing protocols are inappropriate. For example, most current FTP implementations provide no provision for transferring part of a file. One cannot write a new FTP client with this feature because, it will not be useful unless servers all over the world change over to the new protocol. But using our scheme, an FTP client can send a new server application to the remote machine and then use the enhanced FTP protocol.

- **Permits more efficient implementation of applications – e.g., Unix 'cp' on NFS**

  Many operations that involve scanning of huge amounts of data in a remote file and doing little processing on the data can be speeded up considerably if the scanning is done on the server rather than on the client. For example to copy a file from one directory to another within a mounted file system, the usual Unix 'cp' program reads in the file first, and writes out to the target file. Therefore the file is transfered over the network twice. Instead if we could sent a module which does the copying to the remote machine and execute it there, the load on the client becomes virtually zero, and the load on the server will also be reduced because the read and write operations were needed in the normal case also and now we do not need to transmit packets over the network.

  This same performance gain could be achieved for the copy operation by making a small modification to the NFS protocol so that a copy command is defined in the NFS protocol itself. But this approach has the disadvantage that every time you come up with a new operation, (e.g., to determine the number of lines in a file) you will have to incorporate it into the protocol.

- **Makes it easier to administer a large network**

  It is possible to have a *Library Server* of modules, which stores a large number of modules. The servers on the network could simply keep waiting for remote execution requests. The clients could send execution requests to the servers. The servers would fetch modules from the Library Server if the required module in not present in their local cache (Figure 1). Thus to install a new software on all the servers, the system administrator need to just add the module(s) for the new software on the Library Server, and it will automatically become available on all the machines.

- **Flexibility in applications like sendmail**

  Conventional electronic mail can send only text and certain other kinds of data like uuencoded files, etc., for which decoders (uudecode) are generally available. It would be desirable to send all kinds of data like voice, video, etc. (One can do that using MIME.) Currently it is not common because the user sending the mail will not be able to make sure that a decoder for the appropriate data format is available with the receiver. The proposed scheme could provide an alternative solution. One could send the module ID of the mail reader to the receiver of the mail and the actual mail as the parameter of the module. When the receiver wishes to read the mail, the module can be invoked. If the module is not present with the receiver, the scheme will fetch it from the network. Thus the mail application will view the network as a repository of procedures.

Some of the above applications would require the scheme to be very efficient to be useful and be preferred over other approaches. In our scheme, the overhead of invoking a module which is present in the local cache will be equal to that of calling a shared library from a 'C' program.

3 General Outline of the Scheme

This section describes the scheme in brief, design goals, assumptions about the network environment, the caching mechanism and the security issues.

3.1 The Scheme in Brief

The instructions for the Virtual Machine representing the procedure/program to be sent across the network is called a *Module*. We will call the process of conversion of a high-level language code into a module *compilation* and the process of conversion of a module into the machine language of the target machine *translation*.

Every module has associated with it a 128-bit number called the *Module ID*. The module ID is the MD5 sum taken over the entire module. The function computing MD5 [5] sum is a one-way function, meaning that it is computationally infeasible to generate a module with a given module ID.

The scheme consists of four types of entities – *Clients, Translators, Rexec Servers,* and *Library Servers.*

Client: A client is any process which requests the services offered by the scheme. The client could be an application process or an executing module. A client could be local or remote.

Translator: The Translator receives requests from local clients for translation and execution of modules. It fetches the module from the network, translates it and makes it available to the client for execution. The Translators on server machines on the network maintain caches which contain recently used (*translated*) modules.

Rexec Server: The Rexec Server keeps waiting on a socket for client requests. When a *REQUEST_EXEC* message is received, it starts up a new process. The startup code of the newly created process contacts the Translator to obtain the module specified in the *REQUEST_EXEC* message and starts executing it.

Library Server: A Library Server maintains a library of modules which it makes available to Translators or clients upon request. The repository of modules can be either as Virtual Machine code or in the translated form (machine codes for various architectures).

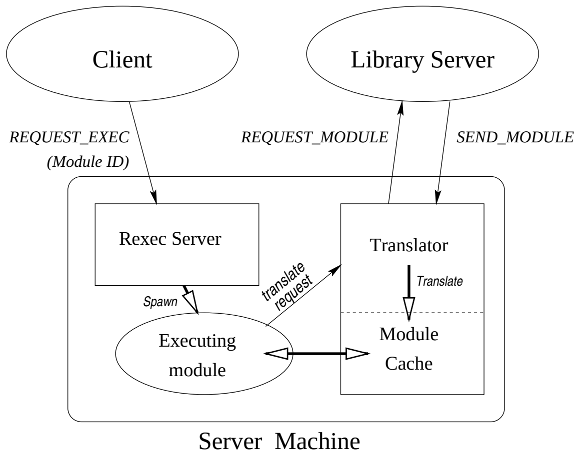

Figure 1. The basic scheme – The client requests the execution of a module on the server by the *REQUEST_EXEC* message. The Rexec server on the server machine spawns a process which during its startup requests the Translator for the module. The Translator searches for the module in its cache, the Library Servers, etc., translates it if required, and makes it available to the new process.

Now we will examine in detail what happens when a client (on machine A) wants to remotely execute the module M on machine B (server) [Figure 1]. The client first sends a *REQUEST_EXEC* message which contains the module ID of M and the parameters for M to the Rexec Server on B asking it to execute it. The message can also optionally contain a list of machines which the Translator on the server should search in case it doesn't have a copy of the module. The Rexec Server, after receiving the request, spawns a process to execute the module. The Rexec Server may grant special privileges (like allowing it to execute with a specified user id) to the spawned process if it can authenticate the client request and if the client is authorized to have those privileges. The new process requests the Translator for the module. The Translator checks in its cache to see if the module is present. If not it asks a Library Server for the module or requests the client or some other machine specified by the client by a *REQUEST_MODULE* message. The requested machine either sends back the module through a *SEND_MODULE* message or refuses to send by a *REQUEST_DENIED* message. The sequence of places where the Translator searches can be controlled by the client by setting various options in the *REQUEST_EXEC* message sent. The Translator can obtain the options and the machines the client specified by contacting the Rexec Server on the same machine.

A module executing on the server can do all the things a normal UNIX process can do provided it has the necessary privileges. Specifically it may also start up other modules, possibly on other machines. The search done for a module invoked in this way is similar to that described above. Thus libraries can be constructed out of a collection of modules which may be used by other modules. This significantly reduces the amount of Virtual Machine code which has to be transferred over the network and translated since a module for a specific purpose can call other commonly available modules to do most of the work, which could be present either in the cache or in one of the Library Servers.

If we were to fork a new process to run a module every time it is called, it would be too expensive to use modules as procedure libraries. Therefore the scheme has methods by which the invoking module or program (client) can ask the Translator to translate the invoked module into the client's own address space directly. This makes overhead involved in invoking such modules equal to that of invoking shared library functions if the invoked module is found in the cache.

3.2 Design Goals

- A module should be completely architecture independent unless the programmer explicitly specifies it to be architecture dependent for efficiency reasons.
- The modules sent need not be large. To make the sending of small modules which run for short periods feasible, the overheads involved in transferring, doing security checks and translating the module should be as small as possible. The Virtual Machine architecture has to be designed in such a way that optimizations can be done while compiling the high-level language into the Virtual Machine code and not during the translation. In short it should be possible to have a single-pass translator which translates without having to look at more than one Virtual Machine instruction at a time.
- The scheme should be extensible while retaining backward compatibility. It is possible that future architectures have an inherently different and efficient way of performing a certain function (e.g., new instructions) which was not anticipated while designing the Virtual Machine architecture. By upgrading the Translator it will be possible to use the new feature. But it may make the translation slower because the Translator has to detect that the feature can be used to replace a number of Virtual Machine instructions. It should be possible to write new modules to substitute old modules which can take advantage of the new feature without compromising on translation speed.
- The scheme should not impose any restrictions on what a module can do. It should be able to access the network, send/execute other modules, access files etc. But a particular machine may impose restrictions.
- Adequate security should be provided. The scheme allows the server to authenticate a module without trusting anybody and to grant privileges to modules after the client provides proper authorization.

3.3 Assumptions About the Environment

The following requirements are to be satisfied by the machines and the network for the scheme to be implemented.

- The scheme assumes that the lower layer protocols provides a reliable packet delivery service with an arbitrarily large packet size. The sequence of delivery of individual packets is not important. This kind of service can easily be built upon TCP.
- The lower layer protocols should have mechanisms for the server to authenticate the packet sent by the client in case special privileges are to be provided to the module. A standard network authentication protocol like Kerberos[2] can be used for this.
- The operating system on the server should have preemptive multitasking capabilities and should provide protected address spaces. It should also enforce access permissions for the various resources provided. Most multiuser operating systems provide all these facilities.

3.4 Module Caching

Module caching at the server reduces network traffic, and reduces server load. Modules are cached by Translators after translation in a ready to execute format. Library servers may cache modules either as Virtual Machine code or as translated code for different architectures.

To illustrate the performance benefits of caching, we shall consider a case in which we want to introduce a new library module. Initially, we put the new library module along with each of the program which uses the module or put it in a Library Server and mention the Library Server's address in the *REQUEST_EXEC* message. The Translator, if the module is not found in the cache, would fetch it from the appropriate place. When the library module is used repeatedly, it would be found in the server's cache or in the Library Server near to the server thus reducing overheads significantly. Hence this mechanism allows to view the network as a repository of program code which are fetched on demand.

This type of caching has certain limitations. If two modules does essentially the same thing, but differs in the implementation, the Translator will not be able to detect it and it will fetch and keep both the modules in the cache if used. But it is expected that a unique implementation of the frequently used modules will be there. A work around would be to configure the translator to fetch and execute other equivalent modules by keeping a list of equivalent modules.

3.5 Security Issues

Three key features by which the scheme provides security are MD5 computed Module IDs, protected address spaces and an authentication mechanism to authenticate clients.

Assume a situation in which a client wants to execute a module on the server. For that it first has to obtain its module ID. This is either provided by some higher level routine or it is obtained by the client itself from the name of the module through some network service like Domain Name Service (DNS) [4].

The Rexec Server could either accept execution requests (*REQUEST_EXEC*) from any client and execute them with the same privileges or could selectively accept connections and give special privileges for certain clients. In the latter case the Rexec Server should be able to authenticate the clients. The Translator downloads the module if not present in its cache. A secure channel is not required for this as the Translator computes the Module ID to verify the authenticity of the module before translation.

Since the Translator can independently verify the authenticity of a module without the help of the client, it can use the same translated module to satisfy any execution request for that module. This would not be possible if the security was ensured by having the server obtain the module from a secure connection specified by the client because two different clients may not want to trust each other to share modules. Therefore it makes caching much more efficient.

The modules are executed in a protected address space. Thus the interface of an executing module to the outside world is through a well defined set of system calls. The security of the system obviously depends on whether the system calls provided are secure enough. The interface to be used and the security measures to be implemented by the system calls are not crystallized yet.

If a local client (a normal program or a module) requests the Translator to execute a module locally, the requested module may execute in the address space of the local client for efficiency reasons. This doesn't create a security hazard because the Module ID of the requested module is provided by the requesting client itself.

4 Module Transfer Protocol

This section discusses the protocol which is used to send modules over the network. The protocol is independent of the lower-layer protocols. A *message* is the unit of transaction between two machines. It is assumed that messages can be transmitted without any loss or errors. There are five kinds of messages.

- *REQUEST_EXEC*: This is the message used to send a request to the remote machine to execute a module. The message contains the following information.
  - *Module ID* of the module to be executed.
  - *Search options*, if any, are used to control the sequence of places where the Translator searches for modules.
  - *Parameters for the module:* These can either be the actual input parameters or a pointer/address of the remote machine from where the parameters can be obtained (e.g., the client which made the execution request). This will be passed on to the new process created by the Rexec Server.
  - *Authentication information*, if required.
- *REQUEST_MODULE*: This message is sent from a Translator to the Library Server or Client to fetch a module in the Virtual Machine format. It may contain authentication information.
- *SEND_MODULE*: This message is send in reply to a *REQUEST_MODULE* message. It contains the requested module. It may also contain search options to control the sequence of places the Translator searches while fetching modules which may be requested by this module.

- *REQUEST_DENIED*: This message is send when a request cannot be granted. It may be in response to a *REQUEST_MODULE* or a *REQUEST_EXEC* message.

- *ERROR*: This message is send as a reply if any kind of error occurs while processing a request.

4.1 Algorithm for Fetching Modules

One needs to specify the sequence of places from where the Translator should request the module. The default sequence of places is:

- Its own module cache.

- A set of Library Servers which are configured when the Translator is started up.

- If a module is requesting the translation, the remote client whose *REQUEST_EXEC* had started up the the requesting module.

This default sequence can be changed by using search options request messages. Through an option, a sequence of addresses can be given which will be searched by the Translator. The option can further specify whether the options will be used only for the immediate children (i.e., modules requested directly by this module) or for all its grand-children also.

5 Virtual Machine Architecture

The Virtual Machine is designed to provide architecture independence, speed of translation, efficiency of the machine code produced and extensibility. The module structure is similar to that of *Executable and Linking format (ELF)*[1, 8, 6] used for binary files by many operating systems.

A module is made up of declarations and instructions. The declarations specify the types of the various variables while instructions correspond to executable code. The declarations and instructions are divided into sections like in ELF.

To achieve architecture independence, rather than viewing memory as a sequence of bytes, the Virtual Machine views it as a place where variables of various datatypes can be stored. *Composite* datatypes can be constructed out of the basic *atomic types* provided, using the *struct*, *union* and *array* constructs (very similar to those in C). Various instructions are provided for manipulating the atomic datatypes and for converting one atomic type to another.

5.1 Datatypes

The design goal was to enable the programmer to provide precisely as much information about the type of the variables as was required by the semantics of the context in which the variable was used. That is, if the programmer wants a sequential counter having at least 16 bits, precisely that information will be conveyed by the datatype of the counter included in the Virtual Machine code. This allows flexibility for the Translator and enables it to produce the most efficient machine code.

5.1.1 Atomic Datatypes

Atomic datatypes are predefined types which are known to all Translators. There are three classes of atomic datatypes supported by the Virtual Machine – integer, floating point, and pointer. The integer and floating point datatype classes includes a number of datatypes of varying sizes so that they cover all reasonable needs of an application.

Integer datatypes: Integer datatypes can be qualified with a *precise* qualifier. If a precise qualifier is present, the variables should take up exactly as much space as the type indicates. If the precise qualifier is not present, memory locations of larger sizes *may* be used to hold these variables and wraparound is not guaranteed. A precise variable may further be qualified by a *network* qualifier in which case the format of storage must be the network byte order.

Floating point datatypes: Floating point datatypes also are available in various sizes and can be qualified by a *precise* or *network* qualifier. The precise type means that it behaves exactly like the IEEE standard types.

Pointer datatypes: Pointer datatypes can be of five types – *global, stack, code, data,* and *heap* pointers. The global pointer can be used to uniquely identify any kind of element in a program. The others are guaranteed to uniquely identify only a specific type of element like code, data, stack, etc. Therefore it is possible that a global pointer needs more bits for storage than the other types. An example of this can be found in the Intel386+ processor where a global pointer *might* include the *segment* and *offset* while any other type would just need the *offset*.

5.1.2 Composite Datatypes

The Virtual Machine supports three composite datatypes – array and struct and union. They are similar to the corresponding elements in C/C++. These three composite datatypes are necessary and sufficient for constructing any memory structure.

5.2 Variables and Register Allocation

A register based design (rather than a stack based one) was chosen because most architectures currently available are register based and we felt that the choice would improve translation speed. Unlike normal compilers, a Virtual Machine compiler will not know how many registers will be available on the real machine. But it will be too costly for the translator to do register allocation to variables itself. Therefore, a Virtual Machine compiler provides "hints" to the Translator specifying which variables are frequently used.

The current version of the Virtual Machine architecture has a large number of virtual registers of each kind. The virtual registers are given priorities. A compiler can specify through the declaration section in which register a variable should go. During translation, the Translator assigns the available real registers to some of the virtual registers according to their priority. This simple mechanism has various drawbacks.

Because the Translator cannot know the flow of the program, all the variables in registers have to be saved before each point where the control can reach from two different locations. This restricts the use of registers considerably. Therefore better schemes have to be worked out.

5.3 Instruction Set

The instruction set should be designed to encompass all the frequently used instructions on currently available architectures. That is, it should be a CISC instruction set. A CISC instruction set would be better due to the following reason. The advantage of a RISC instruction set is that instruction decoding will be faster. This does not apply to the Virtual Machine because the Virtual Machine instructions are compiled into the native instructions. Making the Virtual Machine instruction set CISC will enable efficient translation of the Virtual Machine code into both RISC and CISC native instruction sets. It is very easy for a Translator to convert a CISC instruction to a set of RISC instructions (just a lookup into a table of corresponding instructions will do). In contrast, it will require an intelligent Translator to convert a set of RISC instructions into a CISC instructions because it has to scan quite a few adjacent instructions, rearrange them and then generate the CISC code.

The instruction set was chosen to be orthogonal to reduce complexity. That is, an operation if supported for one datatype should be supported for all datatypes for which it is meaningful.

5.4 Runtime Environment

The Virtual Machine architecture doesn't dictate any kind of runtime environment. But design decisions were taken keeping in mind a runtime environment similar to C/C++. That is, a stack which holds the local variables of a function in invocation frames. But to use the functions in the Virtual Machine API, the runtime stack is required to pass parameters.

A heap, which may be considered part of the runtime environment, is also maintained by the memory allocate/deallocate functions in the Virtual Machine API. Other items include a pointer variable which points to a copy of the parameter field in the *REQUEST_EXEC* message. There is also a pointer to a struct which holds information about what are the privileges of the module, the type of machine on which it is running, operating system type, etc.

5.5 Macros

Macros are analogous to inline functions in C++. During translation, the Translator first translates all the macros into their corresponding native machine code. When the actual module is translated, it substitutes macro calls with the translated instructions. This has the advantage that the Translator has to translate the macro code only once in many cases and that the space required in the module will be less.

But macros were designed mainly to make it easy for modules to take advantage of facilities future processors may offer, while maintaining backward compatibility. Suppose a new processor offer instructions for multimedia operations. Since the Virtual Machine does not have any corresponding instruction, either the Translator has to be smart enough to discover from the code that a set of Virtual Machine instructions could be substituted by a multimedia instruction, or produce inefficient code. But a smart Translator is also a slow one. To overcome this problem macros are used. The Virtual Machine compiler should use a macro whenever it comes across a multimedia operation. It will also generate Virtual Machine code for the macro (corresponding to the multimedia operation) and put it in the module.

The Translator when it sees a macro, puts more effort in optimizing it by collapsing multiple Virtual Machine instructions into the instructions the machine has to offer (in our case multimedia instructions), if possible. If it can do that, then the code generated effectively uses multimedia instructions. If it can't it will simply substitute the normal translation of the macro at the places where the macro is called. Thus we need not modify the standard, yet can make use of the improved capabilities of new processors as though new instructions were added to the Virtual Machine instruction set.

6 Virtual Machine API

The scheme provides an API (Application Program Interface) to the modules. This basic set of functions will be available on all the machines with exactly the same semantics. The API should provide facilities for memory management, network access, file access, process/thread control, access to the module Translator, etc. The API could be provided in two ways.

- *Define a set of standard procedures in the Virtual Machine.* We could define a API procedure call instruction and a set of well defined standard procedures which would be implemented directly by the server machine (for example, through shared libraries). This would be very efficient because the overhead for calling such an API procedure would be almost same as that of calling a C language function. Note that some of the API procedures, like procedures for downloading and translating modules, can only be implemented in this way. It is interesting to observe that there is no essential difference between a Virtual Machine instruction and a procedure implemented this way. We could as well think of the API as an extended instruction set which performs more complicated functions than those in the basic instruction set.
- *Provide the standard API as modules.* The API procedures can be implemented as modules and called just like any other module is called. The API modules can be kept either on the server machine itself or on a Library Server. This method obviously provides more flexibility than the previous method but at the cost of efficiency.

In this scheme we decided to implement the system functions (like those performed by procedures in Sections 2 and 3 of the UNIX manual) and certain others like procedures for downloading and translating modules, which cannot be implemented by the second method, by the first method. This is because it was felt that these procedures are used most often and the loss in flexibility can be tolerated. All other procedures are implemented as modules.

The API procedures should also do extensive security checks so that the security of the whole machine is not compromised. The exact nature of the calls which give access to these facilities are not decided yet.

7 Conclusion

We have presented a scheme to send and receive program modules across the network. As was pointed out in the paper, the advantages of an RPC like scheme which can fetch procedures on demand are promising. We have discussed the scheme in sufficient detail to convince anybody that the scheme is viable. But the implementation efforts are only in their preliminary stages. Only after a working implementation is available, we can verify the various design decisions, and review them, if needed.

A prototype version of the scheme is being implemented currently at *IIT Kanpur*. The initial implementation is being done for Linux 2.0 (a freely available Unix variant) running on an Intel processor. The network used is a TCP/IP network.

The module transfer protocol, security features and module caching are reasonably complete and we don't expect experiments with working implementation would bring about any drastic changes in their structure. But in case of the Virtual Machine architecture, many decisions were taken based on intuition rather than experimental facts and experience with a prototype would bring about changes. This is especially true of the Virtual Machine instruction set, the Virtual Machine API, etc. In summary, the scheme though not complete and stable, is worth further research.

References

[1] *Operating System API Reference: UNIX SVR4.2*, 1992.

[2] J. Kohl and C. Neuman. The kerberos network authentication service (version 5). RFC: 1510, NIC, September 1993.

[3] Sun Microsystems. Remote Procedure Call protocol specification: Ver. 2. RFC: 1057, NIC, June 1988.

[4] P. Mockapetris. Domain names – concepts and facilities. RFC: 1034, NIC, November 1987.

[5] R. Rivest. The MD5 Message-Digest algorithm. RFC: 1321, NIC, April 1992.

[6] R. M. Stallman. Using and porting GNU CC for version 2.6. Technical report, Free Software Foundation, September 1994.

[7] Sun Microsystems, Inc. Java Virtual Machine specifications. Technical report, 1995. Present at http://www.javasoft.com.

[8] *SunOS 5.3 Linker and Libraries Manual*, 1993.