A Scheme for Transfer and Execution of Architecture Independent Procedures

B. Tech Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Bachelor of Technology

by
Manu Thambi

under the guidance of
Dr. Rajat Moona and **Dr. Dheeraj Sanghi**

to the
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur
17th April 1997

Certificate

Certified that the work contained in the report entitled "*A Scheme for Transfer and Execution of Architecture Independent Procedures*", by Mr.*Manu Thambi*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(Dr. Rajat Moona)
Associate Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

17th April 1997

Abstract

Conventional Remote Procedure Call (RPC) mechanisms allows processes to pass arguments to a procedure residing in a remote machine, execute it there and obtain results. I have designed and implemented a scheme which will allow a process to send *the arguments for a procedure to a remote machine for execution. But unlike conventional RPC, the procedure need not reside on the machine on which it is to be executed. The remote machine fetches the procedure as and when required from the network (possibly from the client itself). The procedures (called* modules*) are stored and transfered in an intermediate format, which is independent of any specific architecture or operating system. They are translated on the fly to machine code for execution. MD5 sums over the code of the module is used to identify modules. These can be used by the remote machine to authenticate a module fetched from the network. Security issues are handled by using a MD5 sums and a protected address space provided by the operating system on the remote machine. The scheme also allows a process to fetch a module from the network to be executed locally in its own address space. Caching of modules is used to improve performance.*

This scheme 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 the application running. Whenever a standard protocol is not suitable, the application developer can design a new protocol and create a server program (a module) which can be remotely executed on the server machine. Module caching ensures that the overheads involved are automatically reduced if the protocol becomes widely used.

Acknowledgments

I am extremely thankful to Dr. Dheeraj Sanghi for the continuous encouragement and support which helped me going. I would like to thank Dr. Rajat Moona for the many hours which he spent with me discussing and explaining the many small details of the project. Thanks also goes to Saurabh Sinha for helping me with some coding.

Contents

1 Motivation

Currently the support available for remote execution is very limited. The conventional RPC mechanisms (eg: Sun RPC [Remote Procedure Call, 1988]) helps an application to send arguments for a procedure which resides on a remote machine, execute it there and obtain results. It is the application's responsibility to make sure that the required procedure is present on the remote machine or find a machine where the procedure is present if the machine on which the procedure is executed is irrelevant. 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 for it use the network. This rules out applications like a 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 described above 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 could improve 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 (a Java procedure which can be send over the network for execution) from an HTTP server. The Java Applet received can be either compiled and run locally or interpreted locally. (The currently available Java enabled browsers interpret Java Applets.) Java Applets are send over the network as *Java Byte Codes* written for the *Java Virtual Machine* [Java Virtual Machine, 1995] (defined by Sun), which is simulated on the real machine. Thus the Java Applet code send over the network is independent of the architecture or the operating system 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. (For eg, it cannot do file access or network access to a machine other than the source of the Applet.) It would be 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.

The scheme, described here, 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. The module if not present on the remote machine will automatically be fetched from some machine on the network (which may be the requesting machine also). 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 and operating systems of all the remote machines it is connecting to, 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. The translated modules can be cached so that it saves both the time for transferring the module over the network as well as the time for translating it to machine code. This will make frequently used modules run as efficiently as conventional programs.

2 General Outline of the Scheme

Every module has associated with it a 128bit number called the *Module ID*. The module ID is computed by taking the MD5 sum over the entire module. So changing even a single byte in the module will change it Module ID. The function computing MD5 sum is a one-way function, meaning that it is computationally infeasible to either generate a module with a given module ID or to generate two modules with the same module IDs. Here is a relevant excerpt from the MD5 specifications. [Rivest, 1992]

> *The MD5 message-digest algorithm is simple to implement, and provides a 128 bit "fingerprint" or message digest of a message of arbitrary length. It is conjectured that the difficulty of coming up with two messages having the same message digest is on the order of* 2^{64} *operations, and that the difficulty of coming up with any message having a given message digest is on the order of* 2^{128} *operations. The MD5 algorithm has been carefully scrutinized for weaknesses. It is, however, a relatively new algorithm and further security analysis is of course justified, as is the case with any new proposal of this sort.*

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

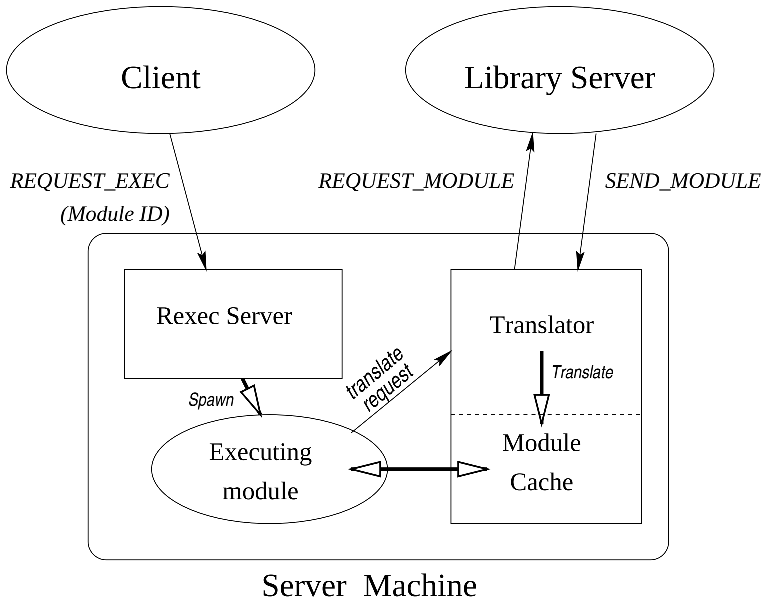

Figure 1: The basic scheme—The client requests the execution of a module on the server by the *REQUEST_EXEC* message. The rexec server on the server machine spawns a process which during its startup requests the translator for the module. The translator searches for the module in its cache, the library servers etc., translates it if required and makes it available to the new process.

could be an application process or an executing module. The client may be running on the same machine as the server (*local client*) or on a remote machine (*remote client*).

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. Therefore the translator need not fetch and translate a module every time it is requested.

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 on the machine on which it is running. 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 in the translated form or as machine codes for various architectures. The server usually contacts a library server which runs on the server machine or machines which can be accessed very fast (for example on the same LAN) from the server.

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 arguments 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 can do all the things a normal UNIX process can do provided it has the necessary privileges. Specifically it may also start up modules on other machines or on the same machine itself. 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 almost equal to that of invoking shared library functions if the invoked module is found in the cache.

3 Module Caching

Module caching at the server reduces network traffic, reduces server load by obviating translation for cached modules and makes the scheme very flexible. Modules are cached by translators after translation in a ready to execute format.

To illustrate the flexibility provided by caching, we shall consider a case in which we want introduce a new library module gradually. Initially, we put the new library module along with each 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 becomes widely used, 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 completely equivalent modules.

In my implementation, caching is realized by using page swapped shared memory which Unix System V provides. When the server is started up, a large chunk of shared memory is allocated. The server maps this shared region into its own address space and translates modules into this space. The local client maps the shared region into its own address space and directly executes the translated code in the shared region. We will discuss this in more detail in section 5.

3.1 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) [Mockapetris, 1987] which the client trusts and whom the client can authenticate using a protocol like Kerberos or RSA.

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 client before executing the module or granting privileges to it. The translator would have to download 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 (network, file system etc.) 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, and we expect that it would require quite a bit of experience with a prototype system.

If a local client (which can be a normal program or a module) makes a request to the translator on its machine to download and execute a module locally, as mentioned earlier, for efficiency reasons the module may be translated into the address space of the local client itself and the requested module executes in the address space of the local client. This doesn't create a security hazard because the Module ID of the requested module is provided by the requesting client itself.

4 Virtual Machine Architecture

The Virtual Machine was designed to provide maximum architecture independence, speed of translation, efficiency of the machine code produced and extendibility. The module structure is similar to that of *Executable and Linking format (ELF)*[API Ref: UNIX SVR4.2, 1992, SunOS 5.3 Manual, 1993, Stallman, 1994] used for binary files by many modern Unix 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.

4.1 Datatypes

The design goal was to enable the programmer/compiler to provide precisely as much information about the type of the variables(storage location) as was required by the semantics of the context in which the variable was used. That is, if the programmer wants just a sequential counter which has at least 16 bits, precisely that information will be present in the type of the counter included in the Virtual Machine code. This allows maximum flexibility for the translator and hence enables the translator to produce the most efficient machine code. Thus a translator on a 32 bit machine can use a 32 bit storage location for the counter and a translator on a 64 bit machine can use a 64 bit storage location for the counter.

4.1.1 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. Each class contain a number of datatypes of varying sizes.

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. That is, if larger values are stored, wrap around occurs. If the precise qualifier is not present, memory locations of larger sizes *may* be used to hold these variables and wrap around is not guaranteed to occur. (eg, a 32 bit location may be used on a 32 bit machine to hold a 16 bit number). A precise variable may further be qualified by a *network* qualifier in which case the format of storage must be a standard one. Thus the integers should be stored only in the network byte order (Big-Endian), in our case. Note that specifying these qualifiers can mean reducing efficiency on certain architectures.

Floating point datatypes: Floating point datatypes also are available in various sizes and can be qualified by a precise or network qualifier.

Pointer datatypes: Pointer datatypes can be of five types—*global* pointer, *stack* pointer, *code* pointer, *data* pointer and *heap pointer*. The global pointer can be used uniquely to 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. For example this can be useful on a i386+ running in 16 bit mode.

4.1.2 Composite Datatypes

The Virtual Machine supports three composite datatypes — array and struct and union. They are similar to the corresponding elements in C/C++.

4.2 Variables and Register Allocation

The translator decides the place where the variables should go. In the Virtual Machine instructions, we refer to variables with logical IDs assigned to them and not through their addresses. For optimization, variables should be selected to be assigned to machine registers depending on their usage. But this takes time and hence the translator cannot do it. The compiler also cannot do register allocation because the number of registers and their properties are different on various architectures. So the compiler provides “hints” to the translator which help the latter to do register allocation. This can be achieved by assigning priorities to variables which indicate their frequency of usage. The exact strategy to be used has not yet been worked out.

4.3 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 and is done only once for every instruction and not once every instruction is executed. 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 normally 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. Therefore making the Virtual Machine instruction set RISC will make it difficult to translate it into CISC machines.

4.4 Macros

Macros are analogous to inline functions in C++. During translation, the translator first compiles 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 included mainly to make it easy for modules to take advantage of facilities future processors may offer. Suppose in the future, almost all the processors offer instructions for say multimedia operations. But 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. But a smart translator is also a slow one. To overcome this problem macros are used. You change the high-level language to Virtual Machine compiler so that it uses a macro whenever it wants to do 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 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.

4.5 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.

5 Implementation

The implementation of the scheme is done for Linux 2.0.X running on an Intel 386+ processor. The network used is a TCP/IP network. The implementation of the scheme can be broadly divided into two parts.

- The Rexec Server, the part of the Translator which does module fetching, security checks, module caching etc. and the Library Server. This part is more or less independent of the actual Virtual Machine architecture.
- The compiler and assembler which converts a high-level language into Virtual Machine code and the part of the Translator which does the conversion from Virtual Machine code to the machine code. This is the part which depends on the actual instructions and structure of the Virtual Machine.

Here we will discuss how the first part is implemented using Unix System V IPC features which Linux supports. Each machine runs three server processes – the *rexec server*, the *translator* and the *library server*. The implementation of the rexec server is very straight forward — the server waits on a well known TCP port for execute requests and forks off a new process when a request arrives.

Central to the implementation of the translator is the module cache. The whole module cache is implemented on a shared memory region which is created when the translator starts up. The size of the shared memory region allocated is the maximum size of the module cache (which is configurable). This doesn't unnecessarily use up memory because Linux allocates actual pages only on demand (ie, when they are used). Since the shared memory is also paged into the disk, are effectively using the disk also for caching. The shared memory region is writable only by the Translator and readable and executable by all processes.

The shared memory region contains two parts — a module index table and the space for the translated modules (see figure 2).

- The **module index table** is a table which has an entry for every module which is either present in the system or is referenced by a module which is present in the system. Whenever the translation of a module is done entries is added to the table for the module being translated and for every module the translated module references. This effectively assigns a unique number (on the server machine), which is the index of the module in the table, to every module present in or used by the system. This unique number if called the *module index.* Each entry in the module table points to the actual location of the module in the cache.
- The translated modules are kept in the second part of the in the shared memory region. Modules are directly executed from this region after mapping them into the address spaces of the local clients.

Other than the normal modules, the second part of the cache contains a special procedure called the *fetch module routine.* All entries in the module index table which doesn't have their corresponding translated modules in the cache point to this routine. We will see the function of this routine a little later.

Every process which needs to use this scheme maps the shared memory region into some part of its address space upon startup. It then opens a Unix Domain Stream Socket with the translator for communicating with the translator. When a process needs to execute a module, it makes a request to the Translator through the Unix socket. When the fetch and translation is complete, the Translator returns the module index to the process. The process indexes into the table and jumps into the module's address. Since the shared memory segment can be mapped any where in the process's address space, the machine code generated by the Translator need to be position independent (i386+ allows you to generate such code – make all jumps relative to the Instruction Pointer). The data areas need a special mechanism which we will discuss shortly.

Consider the case when a module A calls another module B. While A is being translated, the Translator generates code which will call the module B as if it were present in the module cache. When A executes, the module index table entry corresponding to B points to the *fetch module routine*. When A calls B, the *fetch module routine* get control. It then figures out the module called (ie, B) by examining the instruction before the return address and requests the Translator to fetch and translate B. When the Translator is done, it updates the module index table entry for B. The *fetch module routine* jumps to module B using the module index table. Note that the next time the module B is called, there is not even the overhead of checking whether the module B is present.

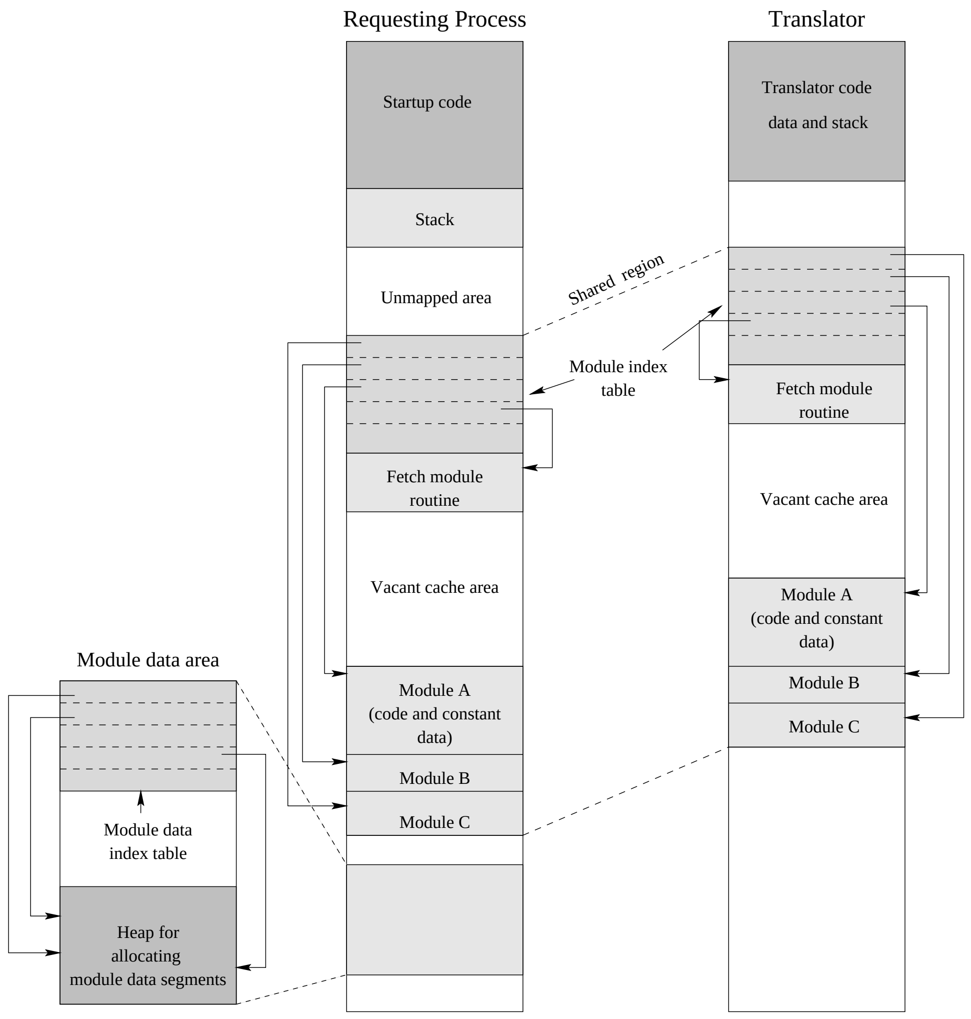

Figure 2: Implementation of the Translator —The address spaces of the local client process and the Translator are shown. The module index table gives the addresses of the modules in the shared memory region, while the module data index table provides the address of the data region of each module.

5.1 Module Data Region

Modules may use data regions other than the stack and heap (Global and static variables in C). The position of the data region of the module cannot be known at translate time because the data regions may be at different locations for different processes using the modules. The work around is as follows.

Every process maintains its own *module data index table*. This is a hash table which maps the module index number of a process to the address of the data region of that module. Every time a module gets control, it hashes (using a simple function like *mod*) into the module data index table. If an entry is not found, ie, if the module is called for the first time from the process, a new section of memory is malloc'ed and the address inserted into the table. During the execution of the module, the address obtained after hashing is maintained in one of the registers. This makes module calls with static data regions slightly inefficient compared to normal C function calls.

Note that the process keeps the Unix Domain Socket open all through its lifetime. Thus when the process terminates (or gets killed), the translator will be able to know about it and will be able to adjust the reference counts etc. on the modules the process had mapped.

The implementation of the library server is straight forward. It simply waits for requests for modules on a socket and supplies them if available. The modules are kept in directories in a tree fashion for efficient access.

I have used a simple Virtual Machine which does not satisfy many of the constraints we mentioned earlier. It allows only integer and pointer arithmetic instructions. An assembler is written which will convert an assembly language Virtual Machine instructions to a Virtual Machine executable. A C/C++ compiler for the Virtual Machine could not be made due to lack of time. The complete description of the Virtual Machine is not attached here due to lack of space.

6 Conclusion and Scope for Future Work

The scheme was implemented so that the server runs with out crashing over extended periods even in the presence of buggy or malicious clients. Extensive checks are there so that the servers do not compromise the security of the system (The rexec-server runs as root so that it can execute setuid() to change the uid of the child process to the appropriate user-id requested by the client.) A simple module to copy a file on a remote machine was written for testing purposes.

Various parts of the implementation have scope for improvement. A compiler from a high-level language to the Virtual Machine code should be written so that reasonably large applications can be written. The instruction set can be revised so as to include floating point instructions. The API provided now is also very minimal. It should include at least most of the system calls available on Unix.

References

[API Ref: UNIX SVR4.2, 1992] API Ref: UNIX SVR4.2 (1992). *Operating System API Reference: UNIX SVR4.2.*

[Hennessy and Patterson, 1994] Hennessy, J. L. and Patterson, D. A. (1994). *Computer Architecture – A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, California, second edition.

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

[Kohl and Neuman, 1993] Kohl, J. and Neuman, C. (1993). The Kerberos Network Authentication Service (Version 5). Request for Comments 1510, Digital Equipment Corporation and Information Sciences Institute.

[Mockapetris, 1987] Mockapetris, P. (1987). Domain Names – Concepts and Facilities. Request for Comments 1034, Information Sciences Institute.

[Nikhil and Arvind, 1989] Nikhil, R. S. and Arvind (1989). Can dataflow subsume von neumann computing ? In *Proceedings of the 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel.*

[Remote Procedure Call, 1988] Remote Procedure Call (1988). Remote Procedure Call protocol specification. Request for Comments 1057, Sun Microsystems, Inc.

[Rivest, 1992] Rivest, R. (1992). The MD5 Message-Digest algorithm. Request for Comments 1321, MIT Laboratory for Computer Science and RSA Data Security, Inc.

[Stallman, 1994] Stallman, R. M. (1994). Using and porting gnu cc for version 2.6. Technical report, Free Software Foundation.

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

[Touch, 1995] Touch, J. (1995). Report on MD5 Performance. Request for Comments 1810, Information Sciences Institute, University of Southern California.