A Remote Execution Model for Mobile Code*

Seung-Hyub JEON†, Min-Hui LIM†, Nonmembers, and Chuck YOO†, Regular Member

SUMMARY The execution model of mobile code inherits from traditional remote execution model such as telnet that needs two conditions. First, the proper program must exist in advance in the remote system. Second, there should be a process in the remote system waiting for requests. Therefore mobile code also bears the same conditions in order to be executed in a remote system. But these conditions constrain an important aspect of mobile code, which is the dynamic extension of system functionality. In this paper we propose a new approach, named Function Message that enables remote execution without these two conditions. Therefore, Function Message makes it easy and natural for mobile codes to extend system functionality dynamically. This paper describes the design of Function Message and implementation on Linux. We measure the overhead of Function Message and verify its usefulness with experimental results. On the ATM network, Function Message can be about five times faster than the traditional remote execution model based on exec().

key words: mobile code, remote execution, dynamic extension of functionality

1. Introduction

The existing model for remote execution such as RPC consists of two parts: client sends requests, and server** receives and processes the requests. This model has two conditions. First, the proper program (e.g. telnet) must exist in advance in the server. If the telnet program does not exist on the server, client cannot request the execution of telnet. If client wants a new service, the proper program (binary code) should be installed on the server before client requests the service. Second, there should be some process (as daemon) waiting for requests from clients to activate the proper program. If there is no process waiting for requests, remote execution does not work.

Remote execution gets more attention due to the introduction of mobile code. The examples of mobile code are Java Servlet [1], Safe-Tcl [2], Omniverse [3]. Servlets are modules that extend request/response-oriented servers in Java Virtual Machine. If client submits a request to execute a servlet, server gets the servlet from clients that are using application protocols such as the HTTP POST and PUT methods, and executes it. Server passes a response to the client in form of HTML file. Safe-Tcl is based on Tcl that is a script language and has been proposed for enabled-mail. In the enabled-mail, a message contains a program to be executed when the recipient reads the enabled-mail. It is executed by an interpreter, and user interaction is allowed. A merit of Omniverse is that it uses well-established programming languages such as C or C++. Then the Omniverse compiler translates into a virtual machine that is safe and language-independent.

The execution model of above mobile codes has the same conditions as remote execution. First, the interpreter or virtual machine must exist in advance in the remote system where the mobile code runs. Second, there should be a daemon or browser waiting for the mobile code. The two conditions of the execution model limit a potential of mobile codes that allows the dynamic extension of remote system functionality. The functionality of a computer system is determined by application programs stored in the system. Although a mobile code is not stored in the remote system, the execution of the mobile code on a remote system has the effect of adding a new functionality (provided by the mobile code) to the remote system. A difference from installing application programs is that remote execution is dynamic because the client can control when and whether it wants the execution of mobile code.

This potential is limited by the two conditions of the current execution model. The first condition requires the existence of Web browser or proper interpreter on the remote system for mobile codes. Before client sends a mobile code, it has to make sure that the sever has proper interpreter. Otherwise, the mobile code will not be able to execute. But in distributed systems, it may not be possible to check the existence in advance. The second condition is that a browser or daemon should be running and waiting for mobile code. Whenever the browser needs to be upgraded with new features, it has to be installed over and over again before the new version is put into service. When there are a large number of servers running, the upgrade of a new browser is very time-consuming. Furthermore, in active networks (the details of active networks are ex-

---

*This work has been supported by Korea Science & Engineering Foundation under contract 961-0909-050-2, 97-01-00-09-01-3 and University S/W Research Center Supporting Project from Korea Ministry Information & Communication.

**Two words-server and remote system are used interchangeably in this paper.
plained in the next section), the remote system can be a router, and it is not reasonable to require that every router has the same version of interpreter.

In this paper, we propose a new execution model that is simple but overcomes the limitations of the current execution model. The key idea is that our model does not require any browser or interpreter. Instead, mobile codes are handled directly by kernel of the remote system. The rationale is that kernel is a common denominator for all computer systems. In other words, every computing device, whether it is a router or a PC, has some form of kernel. If the kernel can handle mobile codes, we believe that the dynamic extension of system functionality is achieved because there is no requirement for browser or interpreter and therefore no need of installation or upgrade of browser or interpreter.

Our model has three characteristics. First, no process or daemon is waiting for mobile code in the server. Second, when a mobile code arrives at the server, the kernel in the server makes a new process out of the mobile code and executes it. Third, the mobile code in our model is executable (binary) code. Since an executable code is transmitted as a message, we call it as function message. In order to dynamically extend the functionality of a server, client compiles and links a source code written in C or C++, makes a function message (FM), and sends it to the server. Therefore, client can send FM any time without checking whether a proper browser of interpreter exists or is running in the server.

The rest of this paper is organized as follows. Section 2 discusses related works. The design and implementation of function message is described in Sect. 3. Section 4 depicts measurement results that analyze the overhead of FM. Section 5 shows limitation of function message, and Sect. 5 concludes the paper.

2. Related Works

The existing model of remote execution is based on Remote Procedure Call (RPC) [5]. The basic idea of RPC is to execute a procedure or handler that exists on a remote system. The sequence of RPC is described in Fig. 1. Client stub sends the identifier of procedure in the remote system and parameters to the procedure, and the server executes the procedure that the client wants to execute and returns a result to the client. Note that procedures must be stored and registered in the remote system before RPC starts. Procedures not registered cannot be used in RPC.

An interesting case of remote execution is active message (AM) [6]. The key idea of AM is that client specifies the address of handler that will be executed in a remote system. Each AM contains the address of handler to be executed and parameters of handler. Figure 2 shows how AM works: (1) client sends a message with the handler address and parameters, (2) when server receives the message, it executes the handler specified in the message in the kernel. AM have something in common with RPC. That is, server should be prepared in advance what functionality client requires, and client should know what functionality server can execute. Server cannot execute arbitrary program, and client can request only functionality that server provides. Therefore RPC and AM are not suited to dynamically extend the functionality system.

Examples of dynamic extension of functionality include active network [7], [8] and extensible operating system. Active network is proposed to allow new technologies or services to be deployed in routers. The computation inside routers has been very limited, e.g., header processing in packet-switched networks and signaling in connection oriented networks. But in active network, the routers are allowed to perform customized computations on the message flowing through them. SPIN [9] is one of extensible operating systems. SPIN allows applications to extend and safely change the operating systems interface and implementation. Extensions written in Modular-3 are loaded into the kernel dynamically at any time. Extension consists of event and handler: event is a message that requests a service, and handler is the procedure that receives the message. Active network and SPIN should guarantee that execution environment does not crash because mobile code from outside is executed in the existing environment. To do this, safe virtual machine or interpreter is used. In active network, operating system running in a router must have an interpreter [10], and SPIN supports extensions only written in Modular-3.
3. Design and Implementation of Function message

3.1 Design

FM consists of a header and a binary code that is fully executable. The operation of FM is as follows: (1) client reads a binary code and creates an FM, (2) FM is transmitted to server, (3) server kernel receives the message and crafts a new process, (4) the process runs the binary code in FM. Figure 3 depicts how FM works. Note that the recipient of FM is not a server process but the server kernel. We call the process that the server kernel crafts out of FM as FM process, and the binary code in FM as FM binary code.

The main issue in designing FM is how to craft a new process from FM in server. Generally, making a new process consists of two steps. The first step is the creation of an address space, and the second is to fill in the address space (e.g. setting program counter and stack pointer to proper values) in order to make the new process run. Traditional kernels like UNIX use a system call (e.g. fork()) to make a new process. The fork system call creates a new address space and copies the address space of the parent process to the new address space. But in FM, there is no parent process to copy to the FM process because it arrives from network. Therefore fork() is not directly applicable. Furthermore, the step to fill in the FM process looks similar to exec() since exec() is used to replace an address space with new one. Unfortunately, exec() is not suitable for FM either because exec() takes a disk path as an argument but the binary code in FM exists on memory, not on disk.

For FM, we designed two new mechanisms called mfork() and mexec(). The purpose of mfork() is to make a new address space, and mexec() is to fill in the address space from the FM binary code in memory and to put the new process into the run queue so that kernel scheduler can schedule. When the server kernel receives an FM, it saves the FM binary code in memory and invokes mfork() and mexec() to make an FM process.

3.2 Implementation

FM is implemented on Linux 2.0.27. To better understand the implementation of mfork(), we first take a look at fork() mechanism in Linux. Here is the description of fork():

(1) Allocate the memory space for a new process.
(2) Copy all the task information from the parent process.
(3) Initialize the new process by setting process id, creation time, etc.
(4) Set the state of the new task to TASK_RUNNING to be scheduled.

The main difference between fork() and mfork() is in step (2) above. Because FM has no parent process, mfork() selects the init daemon process and uses it as if it is the parent process of FM process. Since the init daemon process is the first user process and all the user processes are inherited from the init process, there should be no semantic issue to make the init process as parent. So mfork() creates a dummy process. Then mexec() is invoked to replace the address space of FM process with the FM binary code.

The arguments of mexec() are described in Fig. 4. Note that mexec() takes the memory address of FM binary code instead of disk path as an argument. Here are the detailed steps of mexec():

(1) Analyze the header of FM binary code. This step is required because the header has the information on the binary code format and the address of segments.
(2) Remove unnecessary task information in the FM process. This is because the FM process is copied from the init process by mfork().
(3) Copy the FM binary code to new pages and load these pages to the FM process address space. This means that the addresses of new pages are loaded to page tables so that page faults do not occur when the FM process runs. Traditional exec() does not load page addresses into page tables before it runs. It only stores information about the binary code on disk. When the process runs after exec(), page faults occur and kernel loads the page addresses into page tables when reading the binary code into memory. But in mexec(), the FM binary
code exists only in memory and so the page tables are loaded when the FM process is initially crafted.

4. Allocate new pages that store parameters and environment variables, and maps the pages. The pointers of parameters and environment variables are stored to the user stack. SP (stack pointer) is initialized with the value of next address below these parameters and environment variables. Figure 5 shows the address space of FM process after kernel sets up the user stack.

5. Put the FM process into the run queue after the user stack is initialized. Scheduler selects an FM process, and the FM process runs.

3.3 Comparison

We compare mexec() with exec() in details with an example. Figure 6 depicts exec() and mexec() and shows the layout of an FM binary code that prints "Hello World" on screen. The first part of the layout is the header. In the header, e_ident is the binary code format. The format of binary code in Fig.6 is ELF. We refer to [12]–[14] for the details of ELF. The starting address of execution is in e.entry. The e.offset holds the program header table’s file offset in bytes. The e.phentsize holds the size in bytes of one entry in the file’s program header table. The e.phnum holds the number of entries in the program header table. Physical headers contain information about segments in the binary code. The first physical header represents the information about code segment. The p_type field indicates whether code segment should be mapped during exec(). The address to which a segment is mapped in user space is in p.vaddr, and p_filesz has the segment size. The p_flags field(PF_R) shows that the segment is only possible to read and execute. The p_offset gives the offset from the beginning of the file at which the first byte of segment resides. The p_memsz field gives the number of bytes in the file image of the segment.

Up to this point, mexec() interprets the header in the same way as exec(). Now the difference comes: in exec(), the code and data segments are mapped into the address space (see Sect. 3.2 step(3)), but in mexec(), the segments are copied into memory using anonymous memory mapping mechanism [11]. The reason for this
is to prevent page faults when the FM process runs. If
the pages in the FM process are paged out, since they
are of anonymous memory, they can be loaded back
into memory from the swap area.

Another difference from exec() is that physical
addresses must be used in implementing mfork() and
mexec(). The reason why virtual addresses cannot be
used is as follows. Making an FM process requires ac-
cessing its user space. In a monolithic kernel like Linux,
kernel can use virtual address to access the user space
of a process only when the process is currently running.
When making an FM process, the FM process is not
currently running. In fact, when an FM arrives, the
kernel is running with some other process, and so any
access using virtual address goes to the running process,
not to the FM process being made. Therefore, virtual
addresses are meaningless and physical addresses must
be used for making a FM process. Kernel calls vma-
loc() to allocate physical memory for the FM process
and uses the physical address to access the FM process.

3.4 Protection

FM will not cause kernel to crash because the FM pro-
cess is a user-level process. In other word, if the binary
code in FM is corrupt, the FM process will be ter-
nimated by the protection mechanism (segmentation fault
or bus error) of kernel. Fault protection is very im-
portant in the dynamic extension of system functionality
and FM has an advantage over SPIN because SPIN
needs special checking in order to avoid crashing the
kernel.

4. Experiments

We evaluate the performance of FM with five different
binary codes listed in Table 1. Four different networks
are used: loopback, 10 Mbps Ethernet, fast Ethernet,
155 Mbps ATM.

The experiment environments are shown in Ta-
ble 2. Each binary code is sent with the size of maxi-
mum transfer unit (MTU) of networks.

For each binary code, we measured three metrics
to evaluate the overhead of FM. The first metric is
the time that takes to create an FM process upon the ar-
ival of an FM. This shows the overhead of mfork()
and mexec(). The second metric is the transfer time
between the initial packet arrival and the last packet
arrival on the server. This is to measure the time spent
on the network. The third metric is the overall execu-
tion time of FM, and it is compared with the execution
time of exec().

The purpose is to compare FM where binary codes
arrive from network with exec() that reads from disks.
Machines used in experiments are connected back to
back, and the communication protocol used between
the machines is TCP/IP.

Figure 7 shows the creation time of an FM process.
The results are the average of 100 measurements. The
smallest binary code (code 1) takes about 1.8 msec, and
as the size gets larger, the creation time increases in
proportion of the size. The reason is because the FM
binary code has to be copied to the pages of the FM
process during mexec().

Figure 8 shows the transfer time. The transfer time
of code 5 with 10 Mbps Ethernet is about 2634 ms, and
it is omitted in Fig. 8 because it is much larger than
the others. It is interesting that 155 Mbps ATM has the
best result. We thought that loopback would be the
fastest but ATM is. The reason is that MTU of
loopback is 3580B and ATM has larger MTU (9160B).
When we changed the MTU size of ATM to 3580B,
marked as ATM(3580B) in Fig. 8, loopback becomes
the fastest.

Figure 9 shows the execution time of FM and
exec() on top of ATM. The execution time of exec()
includes the time to read a binary code from disk. The
corresponding part for FM (equivalent of disk read) is
the transfer time. Also since exec() does not include
fork(), the FM execution time includes only mexec(),
not mfork(). Therefore, the execution time of exec() is

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Binary codes and their sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
</tr>
<tr>
<td>code 1</td>
<td>85 KB</td>
</tr>
<tr>
<td>code 2</td>
<td>86 KB</td>
</tr>
<tr>
<td>code 3</td>
<td>107 KB</td>
</tr>
<tr>
<td>code 4</td>
<td>162 KB</td>
</tr>
<tr>
<td>code 5</td>
<td>1 MB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Environment for measuring transfer time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Send side</td>
</tr>
<tr>
<td>Loopback</td>
<td>64 MB RAM, Pentium 180 MHz</td>
</tr>
<tr>
<td>Ethernet</td>
<td>32 MB RAM, Pentium 120 MHz</td>
</tr>
<tr>
<td>ATM</td>
<td>32 MB RAM, Pentium 120 MHz</td>
</tr>
</tbody>
</table>
the sum of disk read followed by exec() plus the run time of the binary code, and the FM execution time is the sum of the transfer time and the time for mexec() plus the binary code run time.

The codes 1 and 5 are artificially-made programs to test two extreme cases. The code 1 has no run time, and therefore the result shows the pure difference between disk reads and network transfer time. The code 5 is a program that has a large code size (with dummy variables) but runs a single line of printf statement. The purpose is to keep the number of disk reads small - the portion of code (single printf) that is actually read from disks is small. The result is that FM is faster than exec() only by 1.3 times because the network transfer time is large due to the big code size.

The codes 2-4 are examples of typical programs (without printf statement). The size of code 2 is similar to that of code 1, but its run time is much longer than other codes. This makes the code 2 (matrix calculation) the worst case for FM in our experiments because the execution time is dominated by the run time. Still, the experiment result shows that the FM execution time is 1.2 times faster than the exec() execution time. The codes 3 and 4 have a balanced combination of the code size and run time. The large difference between FM and exec() in codes 3 and 4 is because the disk read time increases in much faster rate than the network transfer time. Comparing with the code 1, the network transfer time of the codes 3 and 4 increases only a little bit, but the disk read time jumps rapidly. Figure 9 shows that FM is three to five times faster than exec(). This is again due to the fact that disk reads caused by exec() take longer time than network transfer time.

Overall, the experiment results are very encouraging because FM shows superior performance over five different types of codes to exec(). The actual improvement ratio varies depending on the combination of code size and run time. As the network speed gets faster, the network transfer time will be less and less while the disk speed is relatively constant. Therefore, we believe that the merit of remote execution model of FM will increase with faster networks.

5. Limitation

The current implementation of FM supports only homogeneous environments. In the future, we plan to investigate to support heterogeneous environments. A possible approach is to use cross-compiler that generates binary codes for the machine type of server.

6. Conclusion and Future Work

We propose a new approach called Function Message that simplifies the existing model of remote execution. It is simpler than the existing model because our approach does not require browser or daemon but only kernel. The kernel in the remote system crafts a FM process out of the binary code upon the receipt of FM and makes it run. Note that this does not require the installation of new kernel, but “base” kernel can understand FM. Since the FM process runs in user space, it does not add any new security problems to what a mobile code generally has. For example, it does not need to verify an infinite loop or memory violation because an FM process is just another user process so that it does not affect kernel at all. FM is implemented on Linux and tested with five different binary codes. The experiment results show that FM can be five times faster than the traditional execution model based on exec(). Another advantage, we believe, is that FM is not dependent on specific languages, which distinguishes from SPIN or active network. Theoretically, binary codes from any programming language can be used to make FM although our implementation covers only C programming language.

There are several areas that can benefit from FM. A notable one is the remote administration. It is well
known that the administration of a large number of machines is a nightmare. For instance, if a software is to be upgraded, somebody has to install the software manually. But FM can make remote administration easy and simple because remote administration is done by sending FM to remote machines.

References


Min-Hui Lim received the B.S. and M.S degrees in computer science from Korea University, Seoul, Korea, in 1997 and 1999, respectively. Her research interests include high performance network.

Chuck Yoo received the B.S. degree in electronics engineering from in Seoul National University, Seoul, Korea and the M.S. and Ph.D. in computer science in University of Michigan. From 1990 to 1995 he worked as researcher in Sun Microsystems Lab. He is now a Associate Professor in the Department Computer Science, Korea University, Seoul, Korea. His research interests include high performance network and multimedia streaming at thin client.

Seung-Hyub Jeon received the B.S. degree in computer science from Korea University, Seoul, Korea, in 1998. He is currently pursuing for the M.S. degree in computer science in Korea University. His research interests include operating system for distributed and parallel systems.