A proposal for a new active network

Seth Proctor (stp@cs.brown.edu)

October 1, 1998

Abstract

Networks are growing, becoming more complex and used to their capacity. Effort is being put not only into creating more efficient means for network traversal and data retrieval but also building more intelligent ways to monitor how our networks function. At the same time, network technology is moving at a fast enough pace that it's hard to test new network features (protocol stacks, router code, etc) since routers can't always be updated for testing and simulators are difficult to build for custom protocols. Underlying all these issues are increasing concerns about security and QoS guarantees which are sometimes extremely difficult to provide in networks today. In this paper, a new system is proposed which would be built around these concerns and act as a testbed for new ideas, a way of improving network performance and an environment for monitoring networks through internal agents.

1 Introduction

The idea of an active network is relatively new. Instead of keeping the network completely static, as it is now, the elements of the network are programmable, with the goal of making the routing elements dynamic. This means that new routines, protocols, etc. can be fed to the network, thus allowing the previously static environment to change its configuration as needed.

Such an environment is desireable for several reasons. Primarily, this would provide a test bed environment both for new protocols and network simulations, at a level that is otherwise not available. It would also allow a certain amount of work to be pushed out from an end node to the intermediate network, such that there could be fewer bottlenecks due to more intelligent distribution of data. Finally, instead of monitoring, maintaining and updating networks from external nodes, as is done now, this work could be done through the the use of internal agents.

While these all sound like promising applications, there are certain fundamental challenges in building such a system. There must be some way to distribute and execute the new routines. Quality of service guarantees may need to be kept. More important than either of these are the security problems. Allowing arbitrary code to run is obviously a liability. Even code from a trusted source can have flaws, and whatever is executed must not be able to render the resource unusable.

This paper suggests a new approach which provides all the described features, and which protects the host system's ability to continue working reliably. The system consists of several pieces, and so each piece will be discussed separately. Then, they will be described as a single working system.

2 Language

The first important aspect of the new system is the language that is used to program it. In order to make the system useful, the language should be fairly recognizable. It needs to provide all the functionality that a protocol stack might need, but should not be full of extraneous features. It needs to stay small, both for the speed of exectution and the speed of distribution. There are also certain security features that it needs, although while these will appear to the programmer as a part of the language, they are really supported by other features of the system.

2.1 Syntax

With so many languages in existence already, inventing a new syntax would discourage many people from ever using the system. The syntax, therefore, needs to look familiar. In this case, the model is C. In the recent past, new languages (such as Java and Perl) have won many users by providing a familiar C syntax. The syntax, therefore, is C in style, with a few new types added which the supporting operating system uses. Another advantage to using a C syntax is that a simple test library can be provided, which supplies all the custom functionality of the language, and then a program can be built in a native C compiler and tested in a local simulation before testing it on a wider deployment.

Unlike the recent trend in languages, however, this language does not contain any complex notion of an object. The overhead of handling objects and all that comes with them would be prohibitively time/space consuming, and most low level programmers (especially at the network layer) don't want to deal with such speed hits anyway. There are still C structures, and the operating system exports a simple notion of a data vault which provides some encapsulated data sharing.

2.2 Functionality

In terms of the default functionality that the language provides, this is going to be cut down to only the most important set of calls which can be supported anywhere, and which a network programmer needs to work. Many str* (strcpy, strcat, etc) and mem* (memcpy, memmove, etc) calls are supported. Several of the scanf/printf style functions are also provided. Input and output of data on the interfaces is handled through a new set of routines which hook into the operating system.

Many of the ANSI functions, therefore, are provided by default. Any functionality which really isn't needed by the system, however, is cut out. Also, some of the more complicated sets of calls which add overhead to a system (like strtok) are not available.

The language still allows functions to be defined, so that "shipped" pieces of code can come with their own custom routines. In addition, there is a mechanism that the operating system exposes through built-in functionality in the language to export data (such as custom functions) to other protocols. In this manner, new functionality can be added to the system.

2.3 Security

From the standpoint of the programmer, the programming language provides a layer of security to the program. The language gracefully handles illegal access attempts, null pointer dereferencing and array out of bounds indexing. Actually, it's not the language handling these, but a combination of the underlying virtual machine and operating system. This will be described in further detail in the following 2 sections.

3 Virtual Machine

After a program has been written, it is run through the compiler, and this outputs a simple bytecode-like language. The bytecode is distributed throughout the network (with or without accompanying data) and eventually is installed on some node (this process is described in greater depth later). The compiled output is represented in this format so that a platform generic representation of the code can be sent out on the network. Because the program is generic, however, something needs to help it through execution. This is the main duty of the virtual machine.

Positioned between the code being run and the operating system, the VM walks through code, performing certain checks and translations as it goes. There is actually a very fuzzy line between where the VM ends and the OS starts, as the virtual machine is provided through the operating system. For the time being, however, assume that the two are clear, separate identities.

3.1 Speed

In order for the system to be a viable solution, it must perform at reasonable speeds. This means that the virtual machine must be fairly efficient in execution. To this end, when a piece of code first comes in it's processed and loaded into a section of memory where it's ready to be quickly walked through. When a packet arrives which wants to process its data with an installed routine, the code can be quickly executed by moving the input data through a series of commands.

Also, because the virtual machine isn't provided as a separate process, but as an integrated part of the operating system, each routine doesn't require a VM to be started, nor are there several VMs running in the system. This improves performance of the system considerably.

3.2 Functionality

The role that the virtual machine plays should be clear at this point. It is invoked to execute a generic piece of code that has been loaded into the system.The VM knows about state for individual runs, and also knows about both built-in and installed functionality. It is able to accept data, and run this data through a series of routines to completion.

3.3 Security

The VM must also, of course, know about any security restrictions placed on a piece of code, and must obey these requirements. For example, part of its job is to make sure that all memory access are valid, and won't result in crashing the system or overwriting a piece of memory that shouldn't be accessed. Another part of this is watching processor time. No routine should be allowed to render all other routines useless by using up all the processor.

To achieve all this, the VM has a simple set of hooks back to the OS which allow it to query state information, ask for security profiles and request switching. Each routine also stores some information about what it's allowed to do in the system, and the VM will use this to further place restrictions on the routines being executed.

4 Operating System

Coordinating the incoming data, the virtual machine and the threads of execution is the operating system. This is actually a surprising small piece of the system, since it's only doing a few things. It acts as a coordinator, and it provides the pieces of the API which require platform specific functionality. Also, it exposes a small set of administrative features and it juggles some globally exported data.

At one end, the interface to the network is handled. Incoming data is processed, looking for protocol requests, embedded code and security/QoS requirements or credentials. It will install data if the incoming packet was providing a new routine, or send the data to the appropriate routine for processing.

Time sharing and processor monitoring is done by the operating system as well. It will have some idea of how much processor time a thread gets, how much bandwidth it is allowed to use or how quickly its data should be processed.

Based on these factors, the operating system will work with the VM to switch the threads, ensuring fair use for all routines.

The operating system is also doing a number of background tasks such as keeping track of TTL stamps on installed routines and cleaning up old threads and protocols.

5 Complete Framework

When all of the components are combined, the resulting framework provides a complete active network and programming toolkit. This may seem a little complicated, so here's an overview of how the system would be used.

Initially, a program would be written. As an example, lets assume the program is handling data caching. The program would accept any incoming data, look at the request it represents, and then see if the requested data is already being stored at this node. If it is, then the response packet is constructed and returned. If not, then the packet is forwarded on to the next node. This is only about 20 lines or so of code.

Once the program is written, the programmer could compile first using some native C compiler to test it out and make sure that it's working. Then, the program is compiled with the provided compiler, which generates a small piece of bytecode. This is saved to a file for use later. The file represents the program. When this is generated, the program has a unique identifier created for it which is used to identify it later.

When the programmer wants to use the new protocol, they write a program that will write this packet onto the network. Alternatively, if they wish to install the protocol, there are simple utilities included for doing so on a short or long term basis. Still another model would allow for the protocol and data to be sent together, and have the protocol process the data but not install itself on the node.

Assuming a model where the protocol is installed, data can now be sent. In its header are a few flags (provided by helper routines) which identify which protocol to use, any security credentials you wish to provide (certificate, signature, etc) and any QoS requirements.

The data reaches a node, and the operating system reads the data, checks the security credentials (depending on the security policy) and figures out if the protocol is known. If not, then the data is forwarded on. If the protocol specified is known by the operating system, then the data is passed to a thread servicing the correct routine. The data is passed to the previously installed code. Eventually, the queried data is found and returned before reaching the original source node, or the data was never cached, and so the request happens as would normally, and the source machines returns the desired data.

In a more complex system, the data might not be on the current node, but it might be available in the exported data of an adjacent node, and so the first node might go there to get the data, instead of continuing on to the source.

6 Conclusion

The system as described provides all the functionality originally described. It will allow for flexable network configuration, caching and maintenance. Most importantly, pieces of code can safely be introduced into the network to perform a variety of tasks.