CSE 812 Term Project
Socket Shifting

Jason Stredwick, Dehua Hang, Sameer Arora, Grant Birchmeier
{stredwic, hangdehu, arorasam, birchm20}@msu.edu

Instructor: Dr. Philip McKinley
Michigan State University

April 25, 2003

Abstract "Socket shifting" is a client-transparent way to shift a service among a set of servers. It will minimize the overhead and the message exchanges, and it can be used to balance the loads of different servers. In this report, we review and compare existing related work about various ways to provide a socket migration service. Then we introduce our method to implement transparent socket shifting. In our implementation, we use loadable Linux kernel modules (LKM) to override the corresponding kernel functionality, and this kernel modification is only necessary in the client's side. Our LKM functionality exists above the TCP/IP layer; it works by intercepting specific in-stream messages passed by the servers. As directed by these messages, the kernel will provide the socket shifting service for the application level. For our implementation, we decided to modify as little kernel code as possible, to the extent that the servers don't need any kernel modifications. The method does provide reliable and transparent services to the client, but the servers' applications must be written with the assumption that the client has our module installed, and the actual performance needs further study.

1 Introduction and Motivation

Our main goal for this project was to gather experience with Linux kernel programming. The Linux kernel is a very complicated piece of software, and small errors during modification can crash the entire operating system. However, under many circumstances, modifying the kernel is the only way to provide a desired service. Thus, experience with kernel programming will be useful in real world applications. Also, by looking through the source code, we can understand some of the basic OS principles implemented in Linux. Implementing a "socket shifting" service provides an interesting opportunity to work with kernel programming. Socket shifting is a mechanism that allows a server already connected to a client to move its end of the connection to a different server without the client application being aware of the transition.

In client-server configurations where the server is a cluster, a common configuration is for all requests to go through a dispatcher node in the cluster. The dispatcher will determine which other node contains the requested file, and then transfer of that file will be relayed from the node to the dispatcher and then to the client. This creates a notable bottleneck at the dispatcher. Socket shifting could eliminate this bottleneck entirely, as the dispatcher could shift its socket to the other node, allowing the transaction to occur directly between the node and the client.

There are two main classes of solutions for providing such a service: those at application level and those at operating system level. This can definitely be done at the application level by using a migratable socket API, but it will be slower and with more overhead than an implemenation at the OS level. Another benefit of an OS-level implementation is the potential to be transparent to programmers.

The concept of loadable kernel modules (LKM) is a powerful feature in Linux kernel programming that allows modification of the kernel without changing the kernel source code. An LKM is object code that can be dynamically loaded into or removed from a running kernel without rebooting or recompiling the kernel. This will speed up the debugging procedure, since recompiling the whole kernel usually takes an inconvenient amount of time. LKMs are often used to implement device drivers, file system drivers and system calls, and as such have been proven in practice to be a robust OS feature. A LKM is used in our project to override the system call for sockets, which will make the service transparent to the user level.

The remainder of the report is structured as follows. In Section 2 we discuss related work. In Section 3 we describe the details of our test environment, design and implementation. Section 4 shows our results and discussion.

2 Related Work

The primary inspiration for this project was the Cyclone project [1] and [2]. Cyclone is a cluster-based web server system that uses "socket cloning" to shift the page-serving duty from a dispatcher to a server in the cluster. In this system, the dispatcher identifies a server in the cluster that holds the document and uses socket cloning to enable the server to send the pages directly back to the requestor. This technique is useful because it eliminates the "middleman" overhead of the the dispatcher between the the server and the requestor, therefore eliminating a source of bottleneck in the cluster. However, communications *from* the requestor still need to go through the dispatcher to get to the server. We agreed that eliminating this remaining middleman would be a natural direction to proceed. Unfortunately our desire to use this project's source as a starting point failed as our request for the source was denied.

Mockets [7] is an application-level TCP-socket encapsulation providing an API and infrastructure for applications to open network connections and move between hosts without the applications (like mobile agents) themselves having to interrupt, disconnect and reconnect. This project was implemented in Java as a user-level service. The Mocket infrastructure provides a wrapper around the traditional TCP sockets and streams, along with two other processes which together relieve an application from managing a connection during migration.

The Mockets team decided that two sockets were necessary for their implementation. One socket is used for end-point communication and one is used for Mocket control messages. The Mocket control messages suspend, resume and renegotiate the connections between end-point applications. This implementation had unacceptable overhead, therefore, they created a standalone process that would focus the Mocket control system for all applications on a system into a single point. Thus all applications that seek to use Mockets must go through this standalone process, and communication with the remote machine can only occur if they too have the Mocket process. While this approach is similar to ours, as it relieves the client programmer of the need to renegotiate a connection after moving, it is different in that it is implemented at user level while we have modified the kernel. Also, we did not exchange our control messages out-of-band; our implementation intercepts control messages from the transmission when the TCP passes it to the application.

Yau and Lam [5] have implemented a version of Migrating Sockets as an implementation framework for user-level protocols that must provide quality of service guarantees. They implemented this as a library with backward compatibility with Berkeley Sockets so that socket migration would be transparent to the programmer. The internal socket migration is handled using a client/server architecture. All user applications are considered clients and all connections between these clients go through a connection management server. This is different than our project which explicitly avoids the use of a server as a middleman for socket migration. Other differences involve the fact that migration only occurs between a client and the server. Their server will push a socket connection to a client when a connection is one-to-one, and pull the socket back to the server when a connection is a many-to-one or many-to-many. They have a similar idea involving the passing of socket state information as their migration mechanism. We originally planned to do something similar to this, but later decided against it.

MigS [6] is a migrating sockets implementation that aims to provide a kernel-level application-independent socket migration facility to use in building scalable or fault-tolerant systems. Its creators have implemented a session layer End-point Control Protocol to handle operations on communication end-points, and an interface between the session layer and transport layer. The existing interface of the 2.2.16 Linux kernel to the transport layer was deemed adequate. MigS has a much broader goal than our project does and as such has a much more complex implementation.

MIGSOCK [8] is a project that closely parallels our own proposed project. It was a

Master's thesis done to tackle the issue of kernel support for socket migration used in process migration. The support was in the form of a kernel module that completely reimplements TCP in Linux. Using a kernel module made migration transparent to a socket programmer. It did this by using special messages in the kernel to halt and transfer and continue a socket. The user of the socket would never see these messages because they were trapped by the kernel and were never allowed up into user space. We had asked for access to the source, but never received a reply.

The first step of MIGSOCKs was a socket hand-off scenario rather than full-blown process migration. The end goal is almost identical to ours, but the process is different. MIGSOCK put its modification into a module and reimplemented TCP; we leave TCP unaltered and implement our functionality around it. Another similarity is how the socket migration will be demonstrated. The main difference is the service we provide. Both implementations have one program communicate with another who will then pass its connection to a duplicate program on another computer without renegotiation. The last similarity is that we use kernel-level messages to accomplish our task.

Another somewhat related technique is that of network address translation or IP Masquerading [3] [4]. Essentially, the client submits a request to a service, and regardless of how the service sends the data back to the client, the client has performed the proper transaction. The behavior of this service has some similarities to ours, but the specific similarities are few. IP Masquerading is similar in that it maps socket connections in the opposite direction. Instead of mapping the incoming packets to a specific address and port, it maps the address and port of the incoming packets to where the request was originally sent. We want to borrow the basic idea behind the implementation details of IP Masquerading, in that the whole service is invisible to the client.

3 Design

3.1 The Demonstration Application

The scenario created to demonstrate and test the socket shifting design includes three application processes that form a complex echo server: *Client*, *Server1*, and *Server2*. To ease debugging and other logistical issues, all three processes were created on the same machine, thus having the same IP address but each with a different port number; later testing confirmed that our system worked across multple machines. The general operation of the system is as follows. *Client* makes a request to *Server1*, who only has a part of the requested data. After serving its data, *Server1* informs *Client*'s kernel where to connect for the next piece of information by sending a message that will be intercepted at the kernel level. *Client*'s kernel moves the socket connection to *Server2*; *Client* itself is unaware of this transition. *Server2* transmits it's portion of the data and terminates the whole situation by sending a `DONE` message to *Client*, who will terminate the connection.

Discussion of the inner workings of *Client* is important to the following information about socket shifting. *Client* is a simple client application who connects to *Server1* and sends a request for a file by name. Next it goes into a while loop where every iteration blocks on a call to `recv`. On each successful receive, *Client* must process the buffer for multiple messages. These messages are divided using the newline character. Using `strtok` on the newline character, each actual message is processed individually. When the message contains the string `DONE`, *Client* knows that the transaction has completed and exits the while loop to continue its execution.

Our LKM was designed and tested using the Linux 2.4.18 kernel and has functioned on systems running Debian and Slackware distributions.

command start	command string	data appropriate for command string	command end
`7777777=`	**`MOVING_SOCKET=`**	**`0x35670x1e567Myfile.txt=`**	**`7777777=`**

Figure 1: An example of a command message (with whitespace inserted for clarity).

3.2 The Kernel Module

The workhorse allowing our test application to work is our LKM. Its main duty is to parse all incoming data on a socket connection. In order to simplify the parsing task, it invokes generous usage of string functions. The downside of string functions is that arbitrary binary data cannot be handled for two reasons: (1) there is a high probability that the data may contain the equivalent of the null character, and (2) there is a chance that the data may contain the string equivalent of the command message boundaries that have been defined.

Command message boundaries are handled using a somewhat complex message structure. Each command message has a string of seven sevens followed by an equal sign at its beginning and end. Inside each message there is a command string such as `MOVING_SOCKET`, and perhaps data. The command and data are each followed by equal signs. Every command has an associated structure to the accompanying data, which can be binary like the values of a `sockaddr` struct. An example of such a message is in Figure 1.

Intercepting command messages in the sys_socketcall system call is a problematic solution to socket shifting. All socket data for the entire system passes through that system call. There is no good way to distinguish between sockets that want the service and those that do not. The two obvious solutions are sending state information from *Client* to the kernel or hard-coding uids which would be processed in the kernel; neither of these are acceptable. With no good solution, we decided to focus on the test application and ignore extraneous potential problems.

There are possible solutions to handling binary data for socket shifting applications, but none are foolproof. Due to the lack of message boundaries for TCP, a complex message wrapper for every piece of data to be sent would be necessary. A possible format could have a start and end tag that is similar from command messages. The wrapper would surround a value and note the size of the message followed by the message data. An example would be

`666666=`<*number of bytes*>`=`<*data*>`=666666=`

Message parsing is one of the most important aspects of the system. Since these messages are not intended for the receiving application, it can possibly do damage to the system by introducing noise into the received data. Thus it is imperative that these messages be stripped from received data before it gets to the *Client* application. Depending on which command is parsed, the LKM takes different actions. However, the final design only required one message, `MOVING_SOCKET`.

The method for shifting a TCP connection is a simple manipulation of the `task_struct->files_struct->fd` array of open files for that task. By creating a new socket and connecting it to another socket, it accomplishes two tasks. First, it creates all the necessary data structures for a socket. Secondly, initialization of these structures remove the need to identify and manipulate all the TCP-related data members of the socket structures. Due to the difficulty and time required to identify those variables, it became apparent that taking the performance hit from a blocking system call within the kernel was acceptable for this proof-of-concept implementation.

Once a socket is created, the file descriptor returned is the index into the `struct file *` array `fd[]` mentioned above. The first three entries in this array are `stdin`, `stdout`, and `stderr`. When *Client* makes socket function calls, it uses the same file descriptor it received when it created the socket. When the socket system call is invoked, this file descriptor is passed. Knowing this and the file descriptor of the new socket, it was a simple matter to swap the `struct file *` so that now *Client*'s file descriptor uses the new socket. *Client* never knows that it is receiving on a new connection.

4 Results and Discussion

This project has essentially been a proof-of-concept about transparent socket shifting between applications. The implementation is much simpler than other approaches that were examined. More importantly, its modular implementation and application level transparency makes it suitable for many uses like process migration, agents, load balancing, and especially ease of socket programming over a network.

Being a proof-of-concept developed in a short time frame, it does have shortcomings such as a call a function that blocks while inside the kernel. Hence possible future work could start by re-designing the system as a state machine or find the proper data to fill the TCP structures within the module without resorting to a blocking system call. Also, it contains somewhat of an irony in that the client has to have the LKM that is transparent to its application, but the server that's programmed to send the LKM's control messages doesn't need to have it.

This project was a good chance to learn lessons from hands-on kernel-level implementation. We learned that system calls typically expect their data to come from user space, which makes sense conventionally. But LKM support and the corresponding ease of modifying the kernel should be accompanied by a means to pass data to system calls from within the kernel. Also, a Linux kernel crash can have many bad side effects such as corrupting open text files.

Overall, the project gave us ample chance to delve into the kernel and learn more of it's internals - as well as its vagaries.

References

[1] Y.-F. Sit, C.-L. Wang, and F. Lau, “Cyclone: A high-performance cluster-based web server with socket cloning,” *Cluster Computing: The Journal of Networks, Software Tools and Application, Special Issue on Cluster Computing in the Internet.* http://www.csis.hku.hk/˜clwang/projects/cyclone.htm.

[2] Y.-F. Sit, C.-L. Wang, and F. Lau, “Socket cloning for cluster-based web server,” *IEEE Fourth International Conference on Cluster Computing (CLUSTER 2002)*, September 2002. http://www.csis.hku.hk/˜clwang/papers/cluster2002-FNL-socket-cloning.pdf.

[3] D. A. Ranch, *Linux IP Masquerade HOWTO*, January 2003. http://www.ecst.csuchico.edu/˜dranch/LINUX/ipmasq/m-html/ipmasq-HOWTO-m.html.

[4] *Linux IP Masquerade Resource.* http://www.e-infomax.com/ipmasq/.

[5] D. K. Yau and S. S. Lam, “Migrating sockets for networking with quality of service guarantees,” *IEEE International Conference on Network Protocols*, October 1997. http://www.nmsl.cs.ucsb.edu/˜ksarac/icnp/1997/papers/1997-8.pdf.

[6] M. Haungs, R. Pandey, E. Barr, and J. F. Barnes, “Migrating sockets: Bridging the os primitive/internet application gap,” Tech. Rep. CSE-2001-10, University of California, Davis, March 2001. http://pdclab.cs.ucdavis.edu/˜pandey/Teaching/289C/Papers/migs.pdf.

[7] T. S. Mitrovich, K. M. Ford, and N. Suri, “Transparent redirection of network sockets.” http://nomads.coginst.uwf.edu/mockets.pdf.

[8] *MIGSOCK Home Page.* http://www-2.cs.cmu.edu/˜softagents/migsock.html.