Creating an MHOST Table for MNET
An Implementation

Kevin Abas
University of California, Santa Cruz
1156 High Street
Santa Cruz, California 95064-1077
kabas@soe.ucsc.edu

Lu Liu
University of California, Santa Cruz
1156 High Street
Santa Cruz, California 95064-1077
lliu@soe.ucsc.edu

Michael Sevilla
University of California, Santa Cruz
1156 High Street
Santa Cruz, California 95064-1077
msevilla@soe.ucsc.edu

Spencer Sevilla
University of California, Santa Cruz
1156 High Street
Santa Cruz, California 95064-1077
spencer@soe.ucsc.edu

ABSTRACT
This paper discusses details and implementation techniques for an ongoing project called MNET. First, an example socket application is presented and the difficulties of the sockets API is addressed. Then we go into detail as to how MNET is built and how it is added as a linux kernel module with ease. Finally we show other projects that are related to what MNET is trying to fix.

1. INTRODUCTION
In today’s world networking has changed drastically in a way that is counter-intuitive to the original design of point to point conversations. With wireless networking being so widely used, more and more do we see Mobile Ad-hoc Networks (or MANETs) being developed. There is also a lack of effort in attempting to integrate MANETs with traditional IP-based networks. Socket API developers also have problems dealing with multiple socket addresses for a given host, and each application deals with it in their own way. Abstracting the socket API to simplify network application development would benefit groups working on new networking ideas like MANET.

Spencer Sevilla [5] has developed a new socket API family, called AF_MNET, that will lay the groundwork for future MANET development. MNET abstracts the fact that multiple socket addresses belong to a single host, and alleviates the Ochicken-and-eggO problem associated with deployment of network protocols. It is also important to note that MNET is completely backwards compatible and doesn’t effect any of the old socket API functions.

We’ve contributed to the MNET project in a way that makes it deploy-able and functional. In this paper we describe, in more detail, what the motivations are for this new socket API Family and the problems with the current implementation. We then describe how we implemented our table that efficiently stores socket addresses, and how the MNET loadable kernel module is integrated with the current Sockets API.

This work has 3 contributions:

1. Motivation for MHOST
2. MHOST table implementation
3. MHOST table population library

2. TRADITIONAL SOCKET PROGRAMMING
Almost all the network applications we use run in client-server mode. One side of connection, in general the client, has to initiate communication and the other side, the server, will respond. User space application developers use socket APIs to building up connections. Despite the difference in connection and network protocol used, the basic procedure for creating a connection is the same.

1. Create a socket with specified network protocol.
2. Pass a valid IP address and populate an addrinfo struct so it can be used later.
3. Bind the socket to the network address and port. It may be unnecessary for client to bind.
4. Send/receive packets through the socket.

Network applications create, control, and use sockets through system calls such as socket(), bind(), connect(), send(), recv(), etc. We talk about some important system calls in UDP applications. A socket is created by calling socket() which returns a socket file descriptor that can be used later in other socket APIs. Parameters for socket() include
socket family (AF_INET or AF_INET6), socket type (SOCK_STREAM or SOCK_DGRAM), and the protocol used.

Once a socket is created, it can be associated with a port on a local machine by bind() system call. A sockaddr struct containing information about the address (both IP and port) of the machine is passed. After calling bind(), the server is able to listen to incoming traffic at a certain port. The client generally doesn’t need to bind to a certain port unless required.

To send packets, send() (for TCP) or sendto() (for UDP) is used. While in TCP, a connection is created before any data being transmitted, UDP is connectionless. Thus, sendto() requires two more parameters: the address of the destination and the size of the struct storing the address information.

2.1 Example: Chat client

We use a simple UDP based chat program to show the use of traditional sockets. Our chat implementation includes a server program and a client program. The server program maintains a thread to listen to any incoming request and starts a new thread to handle the communication between the client and server. To listen to incoming traffic, a socket is created and bound to a specific port on the local machine. When a request is received, the server creates a new thread. Because UDP is connectionless, the function of the new thread is for the server to send back its immediate response. After that the new thread is closed. When a new packet is received from the same client the server will start a new thread.

The client program is much simpler. All the client does is send messages to and receive responses from the server. But it still requires several socket APIs aforementioned (Figure 3). It creates a socket using AF_INET. Since we are using UDP, connection is unnecessary before calling sendto(). Server IP address and port 8080 are specified. This information is hard-coded into the program. sendto() needs server address to correctly send out packets. In this simple implementation the user needs to explicitly provide the server IP address. The problem of explicitly providing IP address is when the server goes down. The user has to find another IP for the server that works and restart the program again. A better practice is to pass the server hostname and call getaddrinfo() to resolve IP addresses. However, for servers with multiple address, getaddrinfo() will get a list of addrinfo structs and the socket programmer needs to iterate through all of them to determine the one that works.

2.2 Problem of current socket API

Traditional socket APIs use (IP, port) tuple as parameters. This implementation not only requires socket programmers to be aware of the IP addresses of the servers but also violates the separation of layers described in the OSI model. In addition, end users also need to remember IP addresses, which is exceedingly difficult. DNS provides a way to map a string of text to an IP address. It is much easier to remember www.google.com than to remember and use 74.125.224.50. In the case where there are multiple IP addresses for a single host, the advantage of using a string for the hostname is more protrudent. In socket programming, getaddrinfo() is invented to do DNS name lookups for the hostname passed in and automatically populate struct addrinfo. But getaddrinfo() still needs user-specified port number and some relevant information. The client program in Figure 3 is very simple. But the socket programmer needs to specify the socket family and protocol when a socket is created. Thus, any native implementation of a network protocol requires its own address family. When end users provide IP address or a string of hostname, his intend is to talk with the entity identified by the IP address. In order to do that, however, the socket programmer has to use the IP address as routable address when calling sendto(). Thus, IP addresses serve as both the identity of a host and the addressing to the host. In the old days, this is not a problem when one host has only one IP address. But with today's internet applications in mobile smartphones and multihoming, the point-to-point com-

Figure 1: Client-server interaction.

int main(int argc,char**argv) {
  // create a socket
  sockfd=socket(AF_INET,SOCK_DGRAM,0);
  // prepare server socket in struct
  bzero(&servaddr,sizeof(servaddr));
  servaddr.sin_family=AF_INET;
  servaddr.sin_addr.s_addr=inet_addr(argv[1]);
  servaddr.sin_port=htons(8080);
  // send and receive messages
  printf("me> ");
  while (fgets(sendline, 1000,stdin) != NULL) {
    sendto(sockfd,sendline,strlen(sendline),0,
           (struct sockaddr *)&servaddr,
           sizeof(servaddr));
    len=sizeof(servaddr);
    n=recvfrom(sockfd,recvline,1000,0,
               (struct sockaddr *)&servaddr,&len);
    recvline[n]=0;
    printf("server> %s", recvline);
    printf("me> ");
  }
}

Figure 3: A UDP chat client
3. INTERACTING WITH MHOST
The control flow of MHOSTS is shown in Figure 6. The socket programmer would write something like the program test.c in user-space. Our loadable kernel module (LKM) handles everything in kernel space, mainly in the mhost_table.c.

3.1 Client interaction
The left side of the diagram shows a typical program a socket programmer would write to start a connection. In test.c, the user first creates a socket, which is an endpoint for communication - think of this as an outgoing pipe that the programmer can send data through. The socket() system call returns a number, known as the socket file description. This number is used to reference the socket - any time the programmer wants to send a datagram, the socket file description will be used.

Next, the socket programmer translates the “hostname” to an MNET address using mhost_getaddrinfo(), as shown in Figure 2. The socket programmer will use the returned mnet_addr to reference the destination, instead of the traditional (IP, port) tuple. This doesn’t reduce the complexity of socket programming significantly but MHOST can now manage connection loss. If the destination migrates out of range or if it goes down, MHOST can automatically resolve the address to the next address in the MNET address.

3.2 MHOST interaction

In the kernel, the control is routed to our MNET code, written mostly by [5]. Again, we refer to Figure 6 in our discussion of what happens in the operating system.

When the user creates a socket, MHOST creates an empty binding structure, which has 3 components: an id, an l3_head, and a next. Eventually, this binding will be filled with the relevant addresses and the next pointers.

3.3 Prior Work
MNET is implemented as a loadable kernel module (LKM). An LKM is an object file compiled against the library modules of the current Linux kernel (located in /lib/modules/linux-<version>). This executable code is inserted with privileged, kernel-space permissions.

[5]’s code consists of three important files: module_hooks.c, af_mhost.c, and mhost_table.c. module_hooks.c is the entry point for the LKM - when the user inserts the module, the init function immediately calls mhost_init() and launches mhost_table_register(). af_mhost.c “intercepts” all the regular networking system calls (i.e. bind(), socket(), sendto(), etc.) by providing an interface for AF_MHOST. When a system call comes into the system with its family set to AF_MHOST, it is sent to af_mhost.c. Finally, mhost_table.c maps ids to mnet_addr so that the kernel manages IPs and ports, as described below. A small call tree of these interactions is shown in Figure 4.

4. COMPONENTS ADDED TO MNET
This work adds two components to the MNET module that aid in the automatic resolution of addresses and ports via hostnames. The first is the MHOST table, which maps single MNET addresses to multiple, viable addresses. The second is the MHOST Get Address Library, which allows the user to populate the MHOST table at runtime.
4.1 The MHOST Table

The purpose of the MNET table is to store all necessary IP layer addresses for a given a host name. The user will use its mhostaddr which the table then uses to retrieve the correct address the user needs to send to. The table is modified using methods from mhost_table.c and is filled by a new function we define called mhost_getaddrinfo(). The normal getaddrinfo() function is a new implementation of both gethostbyname() and gethostbyaddr(), therefore it handles both name-to-address and service-to-port translations.

This solves the name resolution problem perfectly however it does it in a way that complicates what the user wants to achieve. getaddrinfo() returns a list of addrinfo structs which then a user must have to deal with on his own as described before. mhost_getaddrinfo() also retrieves addrinfo structs however it then extracts what we need from the list and passes that information to the MNET table.

The struct that is extracted is called sockaddr, which is all we need to setup our communication socket. As is defined in sys/socket.h when using any of the socket APIs one must cast their protocol specific struct (e.g. sockaddr_in) to a sockaddr struct pointer. With the MNET table and the LKM the socket app developers don’t need to worry about any of these structs that the normal getaddrinfo() returns, because all of the sockaddr structs are cached and then the sockaddr is chosen correctly for them. Every sockets developer has needed to develop this logic on their own and sometimes in an inefficient manner, if this was abstracted abstracted efficiently then the sockets APIs would be simplified greatly.

The MHOST table also manages and distributes IDs for the hostname to address bindings. This is done in the LKM portion of the MHOST module (module_hooks.c) because this module is persistent for as long as the LKM module is loaded. In a way, this acts as a server; it is always available, responsive, and organized. As such, it is the ideal place for maintaining the ID list - if the ID list were maintained in user-space or in a library, all calls to that program would reset the ID counter.

4.2 MHOST Get Address Library

We implemented a library with a function, mhost_getaddrinfo(“<hostname>”), that allows the kernel to manage the socket addresses for a destination, “<hostname>”. When this function is called, the MHOST table is filled with viable socket addresses and a single address is returned to the user for subsequent connections. These connections take the given address, demux them to the corresponding binding, and send data to an available port and IP automatically. Figure 5 shows the result of using our library to populate the in-kernel MHOST table.

For example, for a connection to www.google.com, the programmer would use the hostname instead of the IP and port number. First, the programmer would create an endpoint for communication with a socket() system call. Then the programmer would call mhost_getaddrinfo(“www.google.com”) to tell the MHOST module where the con-
Starting mhost_getaddrinfo library
Written by M. Sevilla, K. Abas, L. Liu
{msevilla, kabas, liu}@soe.ucsc.edu

Given hostname: www.google.com
− Sending address 0 (74.125.224.50) −−
  family: 2, port: 80, address: 1249763378
− Sending address 1 (74.125.224.51) −−
  family: 2, port: 80, address: 1249763379
− Sending address 2 (74.125.224.52) −−
  family: 2, port: 80, address: 1249763380
− Sending address 3 (74.125.224.48) −−
  family: 2, port: 80, address: 1249763376
− Sending address 4 (74.125.224.49) −−
  family: 2, port: 80, address: 1249763377
− Sending address 5 (0.0.0.0) −−
  family: 10, port: 80, address: 0

− COMPLETED SUCCESSFULLY −− (74.125.224.50)
  * use the returned sockaddr in connections
  family: 2, port: 80, address: 1249763378

Figure 5: When a client uses the MHOST Get Address library, output is printed to the screen to show which IPs and ports the given hostname demuxes to. These addresses are sent into the kernel one at a time. The in-kernel MHOST module returns a single sockaddr structure (just the head of the table), which is used for subsequent connection calls.

Each binding node contains its own linked list called l3_addr which holds a sockaddr pointer and a pointer identifying the short family name for a given sockaddr, for example AF_INET. All normal list modifying functions have been written, however when the table demuxes a mhost_addr it currently just returns the head of a binding’s l3_addr list. In the future a more intelligent l3_addr selection scheme can be implemented which does selection based on the reliability of the sockaddr address.

Each linked list is written in a way that doesn’t create too much overhead which improves upon efficiency greatly. This table will be used by all network applications and a data structure that manages sockaddr structs will never again be implemented incorrectly.

5.2 Modifying the MHOST Table
To populate the MHOST table with viable addresses for the given hostname, we had to implement a function call that could access in-kernel data structures.

5.2.1 Implementing the MHOST Get Address Library
This library, contained in mhost_getaddrinfo.h, allows a process in user-space to populate the MHOST table.

First, the correct character device is opened. The library writes to the character device, which sends the commands and data to the in-kernel MHOST table. This is discussed in great detail in Section 5.2.2.

Next, the library function calls getaddrinfo() with the given hostname and saves the results. Note that this library function needs a hints structure that must be filled in - in our implementation, it is filled with the most generic flags.

The library then iterates through these results and sends them into the kernel for storage using:

ioctl(fd, IOCTL_GETADDRINFO, *casted_addr);

This uses the interface discussed in Section 5.2.2. We had to call usleep() because we were getting synchronization errors - we were sending packets too quickly and the kernel didn’t have time to store them properly. Note that this output has been formatted by the library and that the actual addresses that are sent to the in-kernel MHOST module are sockaddr structures.

Finally, the MHOST module returns an mnet_address, which is just a single sockaddr structure. Again, this is done via ioctl:

ioctl(fd, IOCTL_FINISHED_LOOKUP, mnet_addr);

In this function call, the library sends a pointer of a user-space address (of size sizeof(struct sockaddr structure) to the MHOST module and the MHOST module responds with a pointer to a sockaddr structure. The library outputs a human readable version of the address and returns the mnet_addr to the user.

To provide this functionality to the user, we had to imple-
ment a system-call like function that gives the user the ability to trigger the population of the MHOST table.

5.2.2 Implementing Access to the MHOST Table
When deciding on which methodology to use, we considered complexity, generality, and prototype time. We explored implementing a system call, intercepting a system call, and imitating a system call.

Traditional System Calls
A system call is a function that requests a service from the operating system. System calls are used by user-space programs to access privileged resources in kernel-space. For example, a programmer that wishes to write a file to disk calls a `write()` system call. The parameters are checked by the operating system and if the call passes integrity checks, the file is written to disk.

Our first inclination was to implement our own system call, such as

```c
int mhost_getaddrinfo(
    char *hostname,
    struct sockaddr *mnet_address
);
```

This would call the regular `getaddrinfo()` with the hostname, store the addresses in the MHOST table, and return the single MNET address in the pointer. Implementing a system call like this requires modifying the system call table, creating a library for the system call, and correctly setting the error flags.

Unfortunately, this solution is not simple or general, nor does it accommodate rapid prototyping. This solution is complex, since it requires specific knowledge of the locations and paths of the Linux kernel. It is also very specific to the running Linux kernel and the best case scenario involves writing a sophisticated patch for different versions - even slightly different versions will not behave or compile the same way. Finally, it is tedious and time-consuming since implementation requires re-compiling the kernel. On average, this takes over an hour for a fresh compile and at least 45 minutes for subsequent compilations. This makes it difficult to prototype a working system. We concluded that implementing a system call is not simple and generic, nor does it allow for rapid prototyping.

The second possibility for populating our in-kernel MHOST table was to intercept an existing system call. We would find the system call in the existing `getaddrinfo()` and re-route it to our own MHOST module. To hijack a system call, we need to locate the system call table in kernel memory, copy its entries into the MHOST module, and then search it for the system call that we are looking to overwrite. Then we need to locate the system call assembly instruction so that we can jump into our system call instead of the Linux one.

Although this solution is a little more generic and accommodates quicker prototyping than implementing our own system call, it is incredibly complicated. After Linux 2.6, the developers made it difficult to get your hands on the system call table (it was no longer exported), which means that we would have to find it manually from the `System.map` in the `/boot` directory. Cleaning up the system call after we unload the module would also be tough and unpredictable. Making this solution generic is difficult since kernel memory varies - we would have to make generic traversals and searches to find what we are looking for. Also, the system call parameters are fixed, giving us limited flexibility for extending their functionality. We shied away from this solution because, although it increases prototyping speed and generality slightly, it incurs a huge complexity increase. Besides, this hack undermines the integrity of the system call ideology.

Note that we could also have kicked off functions every time the LKM was loaded. In other words as `init_module()` was called we could fire off a bunch of functions - this is what `init_module()` did in previous demos.

Imitating a System Call
We decided to go with imitating a system call using Input Output ConTroL (`ioctl`) [4]. `ioctl` is a special function that extends the kernel functionality for devices. A device driver is a piece of software that tells the operating system how to operate the physical resource or hardware, as shown in Figure 7. Each device driver in the kernel has an `ioctl` interface for manipulating devices, such as sending and receiving data [2]. User-space processes can call `ioctl` functions to communicate with device drivers. We implemented our system call as an `ioctl` call to our MHOST “device driver”. To do this, we needed to modify the MHOST LKM to accommodate `ioctl` calls and to create a device file in the kernel to write to.

To do this, we modified [5]'s `module_hooks.c`. This compiles into a LKM that, when inserted into the kernel, functions as the MHOST module.

![Figure 7: Device drivers tell the application how to read device files, which represent devices. In MNET, we used `ioctl` and device drivers to modify our MHOST table inside the kernel. Picture taken from [2].](image)

1. The ideology being that that system calls should be solid interfaces for accessing physical resources not vehicles for new functionality.
First, we registered the character device with the kernel. A character device, as opposed to a device file, is a device that participates in character-oriented exchange. To do this, we register the device with a major number. Major numbers represent the type of device (i.e. disk, network card, etc.) and minor numbers enumerate the specific device (i.e. sd1). These numbers are distributed by the kernel to find device drivers. Our LKM prints the register major number in the kernel log.

Next, we created the character device and registered it with the file system. This makes the device available in the /dev directory under the name /dev/my_dev. In this step, we also registered the file operations structure to the device. The file operations structure routes ioctl calls to the corresponding kernel function, as shown in Figure 8. For example, the .open ioctl call can be rerouted so that if the user does something like cat /dev/my_dev, the kernel reroutes control to our corresponding .open call. In our case, we re-routed .unlocked_ioctl so that all ioctl calls are rerouted to our device_ioctl kernel function call.

In device_ioctl, we demux the specific command; we can either call IOCTL_GETADDRINFO or IOCTL_FINISHED_LOOKUP. IOCTL_GETADDRINFO calls a MHOST module function to populate the MOST table and IOCTL_FINISHED_LOOKUP calls a MHOST module function to signify that packets have stopped coming. These functions also have to call special functions to copy values back and forth between user and kernel-space. Once the arguments are converted, control is sent to the various MHOST module functions deeper inside the kernel.

A great majority of our time was wasted because we did not properly manage pointers. When pointers are exchanged across the user/kernel-space boundary, they become jumbled because user-space processes use different memory regions than kernel-space processes. Initially, we were just sending the pointers across as if they were using shared memory - as a result, we got crazy values. The kernel supports copy_to_user and copy_from_user for bulk transfers to accommodate passing pointers across this boundary.

Finally, our modules clean themselves up, unlike [5]'s structures. This is why [5]'s module can only be loaded once. On
the contrary, our structures unregisters the device identifier, destroys the devices, and clear dependencies.

6. RELATED WORK
Service and content-centric networking is upon us. Current works are ambitious and difficult to implement, making MNET all the more attractive.

6.1 CCN
There has been previous work done on how to deal with multiple addresses given to a single host. CCN would make a larger change to the network later and removing the need for IP addresses all together [1]. CCN removes components from the networking layer, like the sockets API, and creates a new networking environment where host-to-host connections are no longer needed. There is also a larger focus on content being transferred rather than where the content is coming from or going to. Rather then thinking of communication as sockets as it is now, CCN allows content to come from multiple locations. Routing can still be handled by the already implemented routing protocols and CCN has a large security advantage.

MNET doesn’t require such a large change and conserves the already implemented reliable IP layer while still abstracting the list of IPs for a given host. Its the fact that the implementation of MNET is so easy that makes it more appealing to the public right now.

6.2 Serval
The Service Access Layer (SAL) [3] is a new architecture for the end host stack and layering model which is based off of service discovery. The routers perform load balancing, server monitoring, control plan reassignments (partitions), and graceful reassignment of connections on an interface (migrate, no down nodes). The approach moves networking towards service-centric networks, especially adept for deployment, scalability, and reliance (churn).

SAL allows applications to talk on service names by modifying the transport layer. Clients request services, the system creates a connection, and the client uses the serviceID to send data. Servers register their service in a pool with other same service servers. The architecture splits service-level control and data, which aids in implementing policy, control, and name-based routing.

Compared to Serval, MNET has a more feasible implementation, is easier to use, and has more generality. Serval makes ambitious stack changes and completely uproots TCP, IP address and ports. As a result, the implementation suffered and the evaluation was weak, especially in regards to the tradeoffs made and the overheads incurred. Serval is also more difficult to use, as new services need to figure out how to register their servers and socket programmers need to become familiar with a new API. MNET keeps the idea of hostnames, destinations, and sources, making it easier for traditional socket programmers to make connections. Finally, Serval is not very general, since applications need to make changes to their code base. For example, to accommodate Serval, the system designer needs to port the Memcached client and server with over 300 lines of code. MNET does not need to change anything at the application level because it is completely backwards compatible.

We are not saying that Serval is a poor idea. We are merely saying that it is too ambitious to realistically integrate into current systems. The ideas are great and enlightening, just a little before their time.

7. CONCLUSION
This work makes three large contributions: we motivate the need for MNET, we implemented an MHOST table, and we exposed functionality for manipulating the MHOST table. The beginning of the paper enumerates difficulties and inadequacies of using the traditional socket API. We then give a brief explanation of MNET and what is already implemented. We presented our implementation of the MHOST table, which keeps a mapping of binding IDs to viable addresses. Finally, we showed how to add functionality to this in-kernel table using the ioctl library. We were able to write a library that allows a user in user-space to populate the kernel-space table without implementing or hijacking system a call. This quarter, we have made great strides and hope that our contributions will be valuable to the MNET team in the future.
8. ACKNOWLEDGMENTS
We’d like to thank Spencer for all his help and attentiveness.

9. REFERENCES