The Minet TCP/IP Stack

Introduction

The Minet TCP/IP Stack is intended to support undergraduate and graduate courses in computer networking that are based on a "students learn by building something real" pedagogical model. The specific goals of Minet are the following:

- Minet enables students to implement a compatible TCP/IP stack that directly controls the Ethernet device.
- Minet lets the instructor control the degree of access students have to the Ethernet device
- Minet enables students to write low-level networking code while still working at the user level with a familiar development environment.
- Minet does not require students to possess knowledge of process or thread control or synchronization.
- Minet works on Linux and should be easily portable to other Unix-like operating systems.
- Minet lets the instructor control the difficulty of an assignment by selective release of C++ classes and Minet modules.

The Minet stack consists of a collection of modules that communicate with each other using, for the most part, fifos or named pipes. Two special modules, which are run as root through the Unix suid mechanism, implement the injection and extraction of raw Ethernet packets. The instructor can modify these modules as necessary to control the packets that students can see. In addition, it is a good idea to use a switched Ethernet network to minimize resource conflicts.

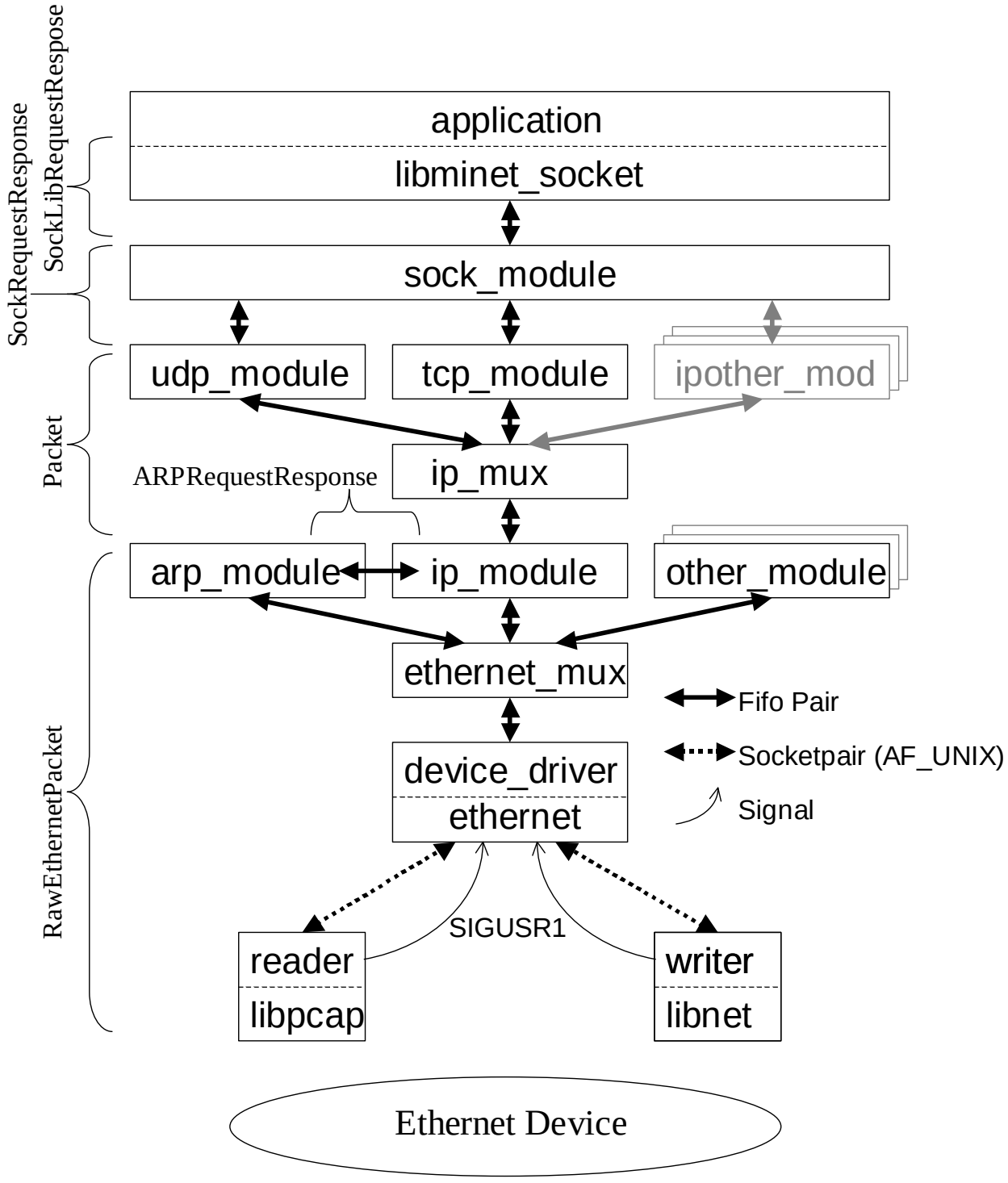
Minet Hardware and Software Requirements

The following hardware is needed to use Minet:

- Intel PC capable of running Linux (ideally at least two such PCs)
- Ethernet card supported by Linux
- Hub or switch (for more than one machine)

The following software is needed to use Minet. Other versions may work as well.

- Red Hat Linux 6.2 (The default kernel will work. If you build your own, be sure to include packet sockets and Berkeley packet filters)
- GCC 2.95.2 19991024 release including STL
- Gnu Make 3.78.1
- libpcap 0.4-19
- libnet 1.0.1-b


SockRequestResponse
SockLibRequestRespose
application
libminet_socket
sock_module
udp_module
tcp_module
ipother_mod
Packet
ARPRequestResponse
ip_mux
arp_module
ip_module
other_module
ethernet_mux
Fifo Pair
Socketpair (AF_UNIX)
Signal
device_driver
ethernet
RawEthernetPacket
SIGUSR1
reader
libpcap
writer
libnet
Ethernet Device

Description of the Minet Stack

The facing page illustrates the standard Minet configuration as of the date of this writing. Minet consists of modules (the labeled boxes), which are separate executables. These modules communicate using paired fifos (heavy bi-directional arrows). The parts of the figure that are in gray are modules that have not yet been completed and are not necessary for basic TCP/IP functionality. A dashed line within a module represents a division between a library and user code. This description is divided into three pieces. First, we describe the data types that Minet provides. Next, we describe the modules, which are implemented using these data types. Finally, we describe the interfaces between these modules.

Data Types

Minet provides a number of data types to simplify development. This section describes the most significant and often used of these. Generally, Minet data types are "serializable" – they support the methods Serialize and Unserialize, which simplify writing them onto a file descriptor or reading them from a file descriptor.

Buffer and TaggedBuffer<T>

The Buffer class provides the basic data buffering mechanism. A TaggedBuffer is simply a Buffer with an associated tag of type T. A buffer is based on an STL crope, which is a string class that is optimized for rapid and easy editing.

EthernetConfig

An EthernetConfig is used to initialize the virtual Ethernet device. It consists of a device number, flags, and a pointer to the interrupt service routine that the virtual device will trap to.

EthernetAddr

This provides a convenient representation of and tools for manipulating 6 byte Ethernet addresses.

RawEthernetPacket

The RawEthernetPacket class is the basic message type for the lower levels of the stack. As the name indicates, it represents a raw Ethernet packet. Like the other message classes in Minet, it provides serialization methods to make instances easy to communicate. RawEthernetPacket::Serialize(const int fd) const writes the packet to a file descriptor, while RawEthernetPacket::Unserialize(const int fd) reads the packet from a file descriptor.

RawEthernetPacketBuffer

This class implements a buffer (or queue) of RawEthernetPackets. This is deprecated.

Header and Trailer

These classes implement packet headers and trailers. They are TaggedBuffers, where the tag represents the type of the header or trailer. Header types include: EthernetHeader, ARPHeader, IPHeader, UDPHeader, and TCPHeader. The only Trailer type is EthernetTrailer. Header and Trailer and their subclasses are carefully designed so that a general Header can be downcast to specific form of header. For example, a Header can be cast to an IPHeader. The representation of IPHeader (a TaggedBuffer) is identical to that of a Header, but it provides tools for conveniently manipulating the raw data of the header.

EthernetHeader and EthernetTrailer

These classes are convenient tools for manipulating Ethernet headers and trailers.

Packet

The Packet class is used to represent all packets except RawEthernetPackets. A Packet consists of a list of packet Headers, a Buffer that represents the payload of the packet, and a list of packet Trailers. The Packet class includes methods for extracting portions of the payload to create new headers and trailers. Packets can easily be constructed from RawEthernetPackets, but the converse is not the case since a Packet may be much larger than a RawEthernetPacket. However, it is possible to extract raw data from the headers, payload, and trailers of a Packet.

PacketQueue

This class implements a queue of Packets. This is deprecated.

ARPPacket

The ARPPacket is a subclass of Packet that interprets the payload as an ARP packet. It provides an example of how to build functionality on top of Packet through subclassing. It is important to note, however, that extend functionality in this manner. Code may also operate on Packets directly, interpreting them based on what headers and trailers are available, cast as appropriate.

ARPRequestResponse

This class represents the request or response for a mapping of IP address to Ethernet address. Generally, these are exchanged only between ip_module and arp_module.

ARPCache

This class maps from Ethernet addresses to IP addresses.b

IPAddress

The IPAddress class is a wrapper around a 32 bit Ipv4 address that provides convenient functionality.

IPOptions

The IPOptions class wraps the raw data of the options that may appear in an Ipv4 Header

IPHeader

An IPHeader provides convenient access to the fields of an IPv4 header, doing the necessary bit-twiddling behind the scenes. Checksums are automatically recomputed as fields change.

UDPHeader

A UDPHeader wraps the raw data of a UDP header in a convenient abstraction. It deals with the bit twiddling necessary to play with the fields of a UDPHeader.

TCPOption

This class contains the options fields of a TCP header.

TCPHeader

This class wraps the raw data of a TCP header in a convenient abstraction, making it easy to manipulate the fields of the header.

Connection

A Connection is a 5-tuple consisting of the source host and port, the destination host and port, and a protocol. The Sock module uses Connections to identify particular flows of data to lower level modules. Connections are used for this purpose even for connection-less protocols such as UDP.

SockRequestResponse

This class represents requests and responses that flow both ways between sock_module and the modules immediately below it

SockLibRequestResponse

This class is used for communicating between sock_module and the libminet_socket.a stubs that the application uses. For example, when the application makes a minet_read call, libminet_socket.a translates this into a SockLibRequestResponse and sends it to sock_module.

Modules

Minet consists of a collection of modules that rendezvous with each other and communicate at run time using fifos. Modules are typically implemented using the facilities provided by the data types described above

Fifo-based Communication model

In order to prevent deadlock, a module must open the fifos it connects to in the following order. First, for each module below it in the figure, from left to right, that it connects to, it should first open the fifo it will read from (the input fifo), and then the fifo it will write to (the output fifo). Next, for each module above it in the figure, from left to right, that it connects to, it should first open the output fifo, and then open the input fifo. Ip_module should treat arp_module as being above it.

Each individual fifo supports messages of only one type. Currently, the following types are available: RawEthernetPacket, Packet, ARPRequestResponse, SockRequestResponse, and SockLibRequestResponse. Each message type supports serialization methods (Serialize/Unserialize) to make it easy to transfer messages across the fifo. The next section describes these types in more detail along with other significant data types.

Structure of a Select-based Module

Minet does not constrain the implementation of a module – as long as an implementation meets its interface requirements, it is a valid implementation. Nonetheless, the expectation is that most modules will be implemented as sequential (single thread) C++ programs based on the select call. This approach has two advantages. First, sequential modules are far easier to debug with standard tools such as gdb. Second, the approach requires only a minimal grasp of operating system concepts. The following pseudocode provides a framework for writing a select-based module.

```
Initialize state of module
Open the fifos of lower-level modules, left to right, input fifo
  first, then output fifo
Open the fifos of higher-level module, left to right, output fifo
  first, then input fifo
while (1) {
     add all fifos to readlist
     select(readlist)
     for each fifo with data to be read {
          unserialize appropriate type from fifo
          update state appropriately
          serialize appropriate types to other fifos
            as necessary
          perform computation and side-effects
     }
}
```

A module may assume that when select indicates that a fifo is readable, an entire data type may be unserialized from the fifo without blocking. In addition, a module may assume that serializing a data type to a fifo will not block if a select call indicates that it is ready for writing. That is, serialization is all-or-nothing. If you can serialize, you can serialize the whole data type.

Virtual Ethernet Device: reader, writer, and the Ethernet Library

The reader and writer modules extract and inject Ethernet packets from and to the network, respectively, and interface to the device_driver module using RawEthernetPackets. The reader and writer modules are special in that they must run as root. The typical way to do this is to provide binaries to the students that have their setuid bit set so that they are run as root regardless of who executes them.

The Ethernet library spawns reader and writer. Unlike the remainder of the modules, reader, writer, and the Ethernet library communicate using Unix domain socket pairs and

signals. The three components form a “virtual Ethernet device”. The virtual Ethernet device operates as a generic DMA block device. It generates virtual interrupts when DMA operations complete and when new packets arrive. These virtual interrupts are derived from signals that reader and writer send to the Ethernet library when new packets arrive or when packets have been sent. The mapping between signals and virtual interrupts is not one-to-one. The Ethernet library assures that one virtual interrupt is delivered for each packet arrival or departure.

device_driver

The device driver module builds a clean interface on top of the virtual Ethernet device. The abstraction is an input queue of RawEthernetPackets that eventually are sent to writer to be injected into the network, and an output queue of RawEthernetPackets that is fed by new packets arriving from reader and which empties into a higher level layer.

ethernet_mux

The Ethernet multiplexor sends incoming RawEthernetPackets to the next appropriate module based on their Ethernet type field. Currently, only two types are recognized: ARP and IP. All other packets are forwarded to other_module, which discards them. The Ethernet multiplexor also accepts outgoing RawEthernetPackets from the ARP, IP and other modules, and forwards them to the device driver for transmission. The figure shows where other modules could be attached for, e.g., IPX or NetBEUI.

arp_module

The ARP module services requests for IP address to Ethernet address mappings, both from the network and from the IP module. It maintains a cache of such mappings. It will only answer requests for its own IP address from the network, but will answer requests for any IP address from the IP module. If the IP module requests an address that is not in the cache, the ARP module will inject an appropriate ARP request into the network.

ip_module

The IP module implements IPv4 functionality. In communicating with the Ethernet multiplexor, it uses RawEthernetPackets. When communicating with higher-level modules, it uses Packets. In such Packets, the IP header has been stripped from the payload and added to the headers section of the Packet. At this point, a Packet will have an Ethernet header, an IP header, payload, and (possibly) an Ethernet trailer.

ip_mux

The IP multiplexor forwards (IP) Packets according to the IP packet type. Currently, only UDP and TCP are recognized. Packets of other types are currently dropped. The figure shows where modules for other types of packets (ICMP, IGMP, etc) would be inserted.

udp_module

The UDP module implements UDP communication, matching (IP) Packets from the IP multiplexor with SockRequestResponse messages from sock_module. The details of this interface are complex and are described below in the Interfaces section.

tcp_module

The TCP module implements TCP communication, matching (IP) Packets from the IP multiplexor with SockRequestResponse messages from sock_module. This interface is complex and described below in the Interfaces section.

sock_module

The Sock module interfaces applications to the modules, such as UDP and TCP that are immediately below the Sock module. The interface to applications is through fifos that communicate with the libminet_socket.a library that is linked to the application. This is a complex interface and is described below. At this point in time, the Sock module supports a single application at a time.

libminet_socket.a

This is a library of stubs that provides the Minet socket interface (separate handout) to user applications. Each stub communicates the call to the Sock module which does the actual work. In addition to Minet, the library can also be initialized to act as a simple wrapper to the kernel socket interface, bypassing the Minet stack. This is convenient for debugging purposes and for use in a networking class that takes a top down approach.

Interfaces

Minet modules generally communicate using pairs of fifos. The protocol by which a module rendezvous with its neighbors on a fifo pair was described earlier. In this section, we will assume that the modules have already connected. The interfaces between modules can then be described in terms of the data types that are exchanged and under what conditions they are exchanged. Each fifo pair carries only a single data type, which simplifies the interfaces considerably. Implementations are simplified by allowing them to assume that serialization or unserialization of a data type will not block if select indicates that the fifo is ready for output or input.

Most of the interfaces are quite simple, consisting merely of Packets or RawEthernetPackets. The more complex interfaces are those between the Sock module and its neighbors. This is because the sock module is responsible for matching application requests and the flow of network data.

Ethernet Library and reader

The Ethernet library spawns reader and communicates with it using a Unix domain socket pair. Reader opens the Ethernet device in promiscuous mode, reads Ethernet packets, filters out packets not bound for this machine (other filters can be added), and Serializes the remaining packets to the Ethernet Library as RawEthernetPackets. After each packet is sent, reader raises SIGUSR1 on the Ethernet Library's process.

Ethernet Library and writer

The Ethernet library spawns writer and communicates with it using a Unix domain socket pair. The Ethernet library sends a packet by serializing a RawEthernetPacket to writer. Writer receives the packet and, at some point in the future, writes it to the network. It then serializes an error code back to the Ethernet Library and raises SIGUSR1 on the Ethernet Library's process.

device_driver and Ethernet Library

The device driver initializes the virtual Ethernet device by calling EthernetStartup with an appropriate EthernetConfig. The EthernetConfig contains the device number to be used and a pointer to the device driver's interrupt service routine (ISR). Once the device is successfully initialized, it will begin calling the ISR when packets arrive or when outgoing packets have been sent. These calls are derived from the SIGUSR1s that reader and writer send to the Ethernet Library. However, they are appropriately massaged so that exactly one call arrives for each received packet or sent packet. There is no telling when these virtual interrupts will occur, so device driver must be very carefully written.

The ISR will be called with a device number and a service type. The service type is either a new packet arrival, which can be read using the EthernetGetNextPacket() function, a DMA completion for an outgoing packet, a DMA failure for an outgoing packet, or an output buffer full failure. The device driver can initiate a DMA to send a packet by calling the EthernetInitiateSend() function.

device_driver and ethernet_mux

The device driver sends newly arrived RawEthernetPackets to the Ethernet multiplexor. Similarly, the Ethernet multiplexor sends outgoing RawEthernetPackets down to the device driver.

ethernet_mux and arp_module, ip_module, other_module, etc.

The Ethernet multiplexor examines incoming RawEthernetPackets from the device driver and routes them to higher level modules based on their Ethernet type field. It forwards RawEthernetPackets arriving from higher-level modules to the device driver. The ARP module responds to ARP requests for the interface's IP address with responses containing the interface's Ethernet address. These addresses are specified through environment variables, which we explain later.

arp_module and ip_module

The ARP and IP modules communicate using ARPRequestResponse objects. When the IP module needs to map an IP address to an Ethernet address, it sends an ARPRequestResponse with the IP address filled in and the REQUEST flag set to the ARP module. If the ARP module finds the mapping in its cache, it fills in the Ethernet address, sets the flag to RESPONSE_OK and sends it back to the IP module. If the mapping is not in the cache, it sets the flag to RESPONSE_UNKNOWN, sends the ARPRequestResponse back to the IP module. As a side effect, it also generates a

RawEthernetPacket containing an ARP request for the IP address and sends it to the Ethernet multiplexor.

ip_module and ip_mux

The IP module communicates with the IP multiplexor using Packets that have an EthernetHeader and an IPHeader. Outgoing Packets arriving from the multiplexor are converted into one or more RawEthernetPackets and forwarded to the Ethernet multiplexor.

ip_mux and udp_module, tcp_module, ipother_module, etc

When the IP multiplexor receives a Packet from the ip_module, it routes it to a higher-level module based on its IP type field. Packets received from a higher level module are forwarded to the ip_module.

udp_module, tcp_module, ipother_module, etc. and sock_module

Communication between the SOCK module and lower-level modules is done using SockRequestResponses, which have serialization features. The model is asynchronous request/response. The sending module sends a SockRequestResponse that encodes its request to the receiving module. In response, the receiving module sends back a SockRequestResponse that encodes the status of the last action. A SockRequestResponse contains a request type, a Connection , a Buffer containing data, a byte count, and an error code.

Request/Response Ordering

Note that it is possible for both modules to send a request first. This is not a race condition - in principle, a module should be able to handle responses asynchronously as they arrive provided that the responses are returned in the same order that their requests are made. In other words, there must be a total order on requests and a total order on responses, but there may be only a partial order on requests and responses together. This means that responses do not need (and do not have) any field to indicate which request they match to.

Connection Matching

A fundamental abstraction is that of a Connection, which is used even for connection-less protocols such as UDP. The Connection structure encodes the endpoints of the communication (ipaddresses and ports) as well as its protocol. One of the endpoints may be unbound if it will be supplied later (for example, a passive TCP open). The Sock module and lower-level modules identify particular flows of data by using a Connection. For example, the Sock module presents sockets to the application as integer file descriptors. Internally, it maintains a file descriptor to Connection mapping. When the Sock forwards, for example, an ACCEPT request, to the TCP module, it uses a Connection to identify the connection to the TCP module. The TCP module maps the Connection to its internal representation as appropriate.

sock_module to udp_module
Here are the meanings of the different types of SockRequestResponses that can be sent from Sock module to the UDP module:

- CONNECT: active open to remote. The UDP module ignores this. A STATUS with the same connection, 0 bytes, and an error is returned. The Sock module must manage connect requests to UDP addresses.
- ACCEPT: passive open from remote. The UDP module ignores this. The same behavior as CONNECT is required.
- WRITE: send UDP packet. Connection source is the local host and port, the connection destination is the remote host and port, and the protocol is UDP. The data Buffer contains data to be sent (if data is larger than the maximum size UDP packet that the UDP module can sent, then only the first maximum size bytes are sent. The byte count and error code are ignored. The response is a STATUS with the same connection, no data, the number of bytes actually sent, and the error code. One WRITE generates one UDP packet.
- FORWARD: forward matching packets. The connection represents the connection to match on. The local and remote addresses may be wildcards (IPADDR_ANY, PORT_ANY). Received matching packets will be forwarded to the Sock module as WRITEs. The response is a STATUS with the same connection, no data, zero bytes, and an error code.
- CLOSE: close connection. Connection represents the connection to match on. If there is a matching FORWARD request, this will remove it. Otherwise it is an error. A STATUS with the same connection, zero bytes, and the error is returned.
- STATUS: status update. This should be sent in response to UDP WRITEs. The connection should match that in the WRITE. The error code is required but ignored. The byte count is the number of bytes that were actually received.

Udp_module to sock_module
Here are the meanings of the different types of SockRequestResponses that can be sent from the UDP module to the Sock module:

- WRITE: new data is available. The connection is fully bound, giving both endpoints and the protocol, the data Buffer contains the data in the packet, and the byte count and the error code are undefined. In response, the Sock module should send a STATUS stating how many bytes were read. Note that UDP will not buffer data.
- STATUS: status update. This is sent in response to Sock module requests as noted above.

sock_module to tcp_module
Here are the meanings of the different types of SockRequestResponses that can be sent from Sock module to the UDP module:

- CONNECT: active open to remote. The connection should be fully bound. The data, byte count, and error code fields are ignored. The TCP module will begin

the active open and immediately return a STATUS with the same connection, no data, no byte count and the error code. After the connection has been fully established or has failed, it will return a WRITE with zero bytes. If the connection could not be established, the error code will be non-zero. The Sock module must manage binding on CONNECT requests.

- ACCEPT: passive open from remote. The connection should be fully bound on the local side and unbound on the remote side. The data, bytes count, and error fields are ignored. The TCP module will do the passive open and immediately return a STATUS with only the error code set. Whenever a connection arrives, the TCP module will accept it and send a zero byte WRITE with the fully bound connection.
- WRITE: send TCP data. The connection source is the local host and port, the connection destination is the remote host and port, and the protocol is TCP. The connection must refer to the result of a previously successful ACCEPT or CONNECT request. The data Buffer contains the data to be sent, while the byte count and error fields are ignored. The response is a STATUS with the same connection, no data, the number of bytes actually queued by the TCP module, and the error code. One WRITE may generate multiple TCP segments. It is the responsibility of the Sock module or of the application to deal with WRITEs that actually write fewer than the required number of bytes.
- FORWARD: forward matching packets. The TCP module ignores this message. A zero error STATUS will be returned.
- CLOSE: close connection. The connection represents the connection to match on and all other fields are ignored. If there is a matching connection, this will close it. Otherwise it is an error. A STATUS with the same connection and an error code will be returned.
- STATUS: status update. This should be sent in response to TCP WRITEs. The connection should match that in the WRITE. It is important that the byte count actually reflects the number of bytes read from the WRITE. The TCP module will resend the remaining bytes at some point in the future.

tcp_module to sock_module

Here are the meanings of the different types of SockRequestResponses that can be sent from the UDP module to the Sock module:

- WRITE: new data on a connection. The connection will be fully bound, the data buffer will contain the data, and the other fields should be ignored. In response, the Sock module should send a STATUS with the same connection and the number of bytes it actually accepted. The TCP module will resend data that has not yet been accepted.
- STATUS: status update. This is sent in response to various connection requests as described above.

sock_module to libminet_socket.a

In progress – Kevin Dill

Setup and Configuration

Minet is supplied with a Makefile and various scripts to simplify building and using it.

Building Minet

You must have the software environment described on the first page. This is the only environment on which Minet has been built and tested. To build Minet do the following:

1. Unpack the Minet tar file
2. cd minet
3. touch .dependencies
4. make depend clean all

You should now have the following items:

- libminet.a – general library of Minet data types
- libminet_socket.a – stub library for Minet applications
- modules: reader, writer, device_driver, ethernet_mux, arp_module, ip_module, other_module, ip_mux, udp_module, tcp_module, sock_module (Note: you may have a different list of modules depending on your instructor. Furthermore, some modules, such as reader and writer, may be supplied to you in binary form only.)
- app – a sample application

If you are building reader and writer, you must set their permissions appropriately. The script fixup.sh, WHEN RUN AS ROOT, will set the permissions correctly.

All Minet compile-time configuration options live in the file config.h. Because most options are run-time, you will probably not have to ever change this file.

Configuring Minet

To configure Minet, edit the file setup.sh and then source it. It is important that you SOURCE the script and not simply run it because it sets a number of environment variables. If you run it, these variables will be set only in the child shell and not your shell, and thus will not be inherited by other programs you run. In addition to setting up your environment, setup.sh will also create fifos (in the directory ./fifos) for you if they are needed. Depending on your instructor, it may also set up reader and writer binaries.

The following describes the environment variables that setup.sh sets:

- MINET_IPADDR – the IP address assigned to the virtual Ethernet device.
- MINET_ETHERNETDEVICE – the physical Ethernet device that will be used by Minet. This is typically “eth0”
- MINET_ETHERNETADDR – the Ethernet address of the actual Ethernet card. You can run /sbin/ifconfig to find out what this should be
- MINET_READER – the path of the reader binary that will be used
- MINET_WRITER – the path of the writer binary that will be used

- MINET_READERBUFFER – the size of the input buffer on the virtual Ethernet device. Deprecated.
- MINET_WRITERBUFFER – the size of the output buffer on the virtual Ethernet device. Deprecated.
- MINET_DEBUGLEVEL – the debug level for debugging printfs. Zero denotes no debugging output. Higher debug levels result in more debugging output
- MINET_DISPLAY – how Minet should display output when run. "xterm" means each module (except for reader and writer) runs in its own xterm window. This is the usual mode of operation. "gdb" means that each module is run in a separate xterm under the gdb debugger. "log" means that each module is run in the background with its output going to log files. For example, ip_module would produce ip_module.stdout.log and ip_module.stderr.log.
- MINET_MSS – TCP maximum segment size
- MINET_MIP – Maximum IP packet size
- MINET_MTU – Maximum Transmissible Unit of Ethernet device
- MINET_FRAGMENTATION – enables IP fragmentation support if defined
- MINET_CONGESTION – enables TCP congestion control if defined
- MINET_ROUTING – enables IP module routing if defined

This list will very likely change as Minet becomes more robust

Running Minet

Make sure that you have SOURCED setup.sh. The following assumes further that MINET_DISPLAY="xterm". To start Minet, run "go.sh". You should very quickly have at least 10 xterms on your display, each running a module. If some of the xterms immediately disappear, make sure that the device_driver xterm is not showing an error. It is essential that device_driver be able to spawn reader and writer successfully. Check that the environment variables described above are set reasonably.

What is displayed in the xterms depends on network traffic and what the application is trying to do. The default application does nothing but sit and print "la la la".

To stop Minet, run "stop.sh". The xterms should all disappear.

Security Concerns

Minet encapsulates access to the network in the reader and writer modules. It is critical that reader and writer binaries have appropriate protections. We strongly recommend that they be supplied only in binary form, be owned by root, have root-only execute permissions, and have their setuid bits set so that ordinary users can run them.

Both reader and writer are quite simple and offer significant opportunity for filtering. Reader filtering is particularly simple because it already supplies a Berkeley packet filter program to libpcap.

We also recommend that student traffic be constrained to the 10.0.0.0 subnet so that it doesn't leak to the outside world. It is also a good idea to have the students work on a switched Ethernet network to minimize resource conflicts.