CSCI367
Spring 2014
First Programming Assignment
Unidirectional Data Flow Through Piggy

We are going to build the final version of this term's project by building a sequence of programs each of which builds on the previous. The first step was what you did Friday 4/42014 (3rd class meeting) in just getting a simple "echo" client and server programs working for your choice of implementation language from the samples I provided. This was an ungraded exercise just to get the ball rolling and to ensure that you understand the mechanics of how to operate the computers to get the programs talking to each other across the network. If you are uncomfortable with any aspect of this lab please make sure you see me early in this week get you on track. Now we begin a sequence of graded programming exercises which ultimately lead up to the finished product for this terms project. Each step along the way will have a due date. All submissions will be made through Canvas. I will download the set of submissions for the class and run a set of tests on them. You must meet with me to see the tests executed against your submission. In this manner, yo will receive constant feedback along the way as to how your project is shaping up. The meeting to review your submission can take place in our Friday lab sessions or during my office hours on Mondays if need be. You may also make an appointment outside of these times but make sure it happens as early after the Friday due date as you can in order to keep up with the current week's assignment.

The first day of class in reviewing the syllabus, I gave you an indication of where we are headed as our final product for this term's project in the syllabus. By the end of the term we will have a very capable network data stream filtering and analysis tool called piggy. It will require us to build successive upon a number of incremental steps in order to get there. The discussion in the syllabus was written from the assumption there would be two phases, which is the way the project worked during the Winter term. I have decided this term to break the task down into even smaller increments. We will have a sequence of many steps leading up to the final version. The exact number of steps that will be required is not completely known at this point. Since we have only 10 weeks to get to the finish line the most conceivable would be 9. I cannot imagine that it will require that many. My guess at this point would be more on the order of 5. If that requires that the last step of the build of the project be due during "dead week" then so be it. Consider yourself as having been notified that the last step in the construction of the project may end up being due during the last week of the term. The syllabus indicated a grading scheme of: Exams 45%; Projects 45%; Protocol Team Participation 10%. However many programming steps we end up with along the way will all go into the Projects category. In the past I have tried to allow some credit to go towards our participation in a protocol design group as I believe the experience of working with others in the design implementation and revision of an application protocol specification is an important learning experience. I will try to retain this component of the grading scheme so that you as a class can participate in the later stages of the final design for the project. Unfortunately, given our severe time constraints with a 10 week term, it is not possible for the class to participate in the design phase of the early assignments. In the interest of efficient use of our time I have provided the complete design for the initial pieces. This document describes the first of these graded programming components known as piggy1.

Piggy-1

Recall that Piggy is a program that is designed to sit in between two programs (perhaps a client and a server) and display and manipulate the data streams flowing between them. As a first step toward building this we are going to construct a program that

- Optionally accepts a connection as specified on the command line
  and
- Optionally makes a connection as specified on the command line

To make the first program simple we are only going to deal with data flowing in one direction "left to right." We are going to display the data that comes in from the left side to the screen and also write it to the right if there is one.

The command line parameters the program accepts are as follows.

Command line parameters can appear in any order All command line parameters will assume we are dealing with the bash shell in Linux. For all commands if the right side port address is not specified you should use your "primary" port address assigned to you for the course, 367xx, where xx is the book number you have been given.

Specifying Address of the left and right sides

`-laddr value`

This is used to specify what address are valid when accepting a connection from "the left side." Value is either and IP address in dotted decimal notation or a DNS name or the character *****. Note that ***** must be quoted as in "*" in order to prevent the shell from thinking it should expand it to a list of the file names in the current directory.

`-raddr`

This is used to specify the address of the right side, i.e. the node we should connect to. Value is either and IP address in dotted decimal notation or a DNS name. Note that unlike the laddr option "*" is never valid for an raddr. Also, note that there is no default address for the right side. An raddr must always be specified unless the -noright option is given.

`-noleft`

This indicates that there will be no left side connection. This is useful to create the "head" of a chain of piggy processes from which we want to generate a stream of data. If `-noright` is given then any `-laddr` option is ignored.

`-noright`

This indicates that there will be no right side that piggy connects to. This is useful to create the tail of a chain of piggy processes. If `-noright` is given then any `-raddr` option is ignored.

Specifying a port address

`-lacctport value`

This indicates what source port will be accepted. Value is a valid port address which in the case is any 16-bit unsigned integer value, i.e. 0..65535. It may also be the wild-card * indicating any port is acceptable. The default is to accept any valid source port address.

`-luseport value`

This option is used to indicate what port should be used for the left side connection. The default is to use your primary port you have been given for the course, i.e. 367xx where xx is the number of the text you have been given.

Example invocations and their interpretation

```
piggy1 -laddr 140.160.140.5 -raddr 140.160.140.5
```

This command would cause piggy to accept a connect on the left side only if it originates from a computer with the IP address 140.160.140.5. Additionally the program would try to make a connection to the node with IP address 140.160.140.5. The incoming connection would accept any source port. The outgoing connection would use the default port address.

```
piggy1 -laddr bucky.cs.wwu.edu -raddr 140.160.140.70
```

This would cause piggy to accept a connection form the node with DNS name bucky.cs.wwu.edu and connect to a node with an IP address of 140.160.140.70. The incoming connection would accept any source port. The outgoing connection would use the default port address.

```
piggy1 -laddr "*"  -raddr bucky.cs.wwu.edu
piggy1 -raddr bucky
```

Both of these commands would accept a connection on the "left side" from any IP address. Note that `-laddr "*"` is never needed to be specified as this is the default action. Note ,however, that if the `-noleft` option is given then any `-laddr` option is ignored so strictly speaking * is the default in the absence or a `-noleft` option. Piggy would make a connection to a node with the DNS name bucky. Assuming that the default DNS domain for the computer executing the command is cs.wwu.edu each command would attempt to connect to bucky.cs.wwu.edu. The incoming connection would accept any source port. The outgoing connection would use the default port address.

```
piggy1
piggy1 -noleft -noright
```

Both of these commands would produce an error message indicating that you must have either a left or a right side address.

Piggy1 -luseport 26799 -lacctport 36799 -raddr bucky

This command would accept a left side connection from any IP address as long as the source port address from the connecting machine was 36799. It would use port address 26799 as its local port

address for the left side connection. It would try to make a connection to the node named `bucky` on the default DNS domain.

Keyboard Input

In this first program we are intentionally ignoring any issues related to blocking and the need to deal with multiple streams of input flowing into the program and through the program. In the event that -noleft is given your program will simply read what is typed at the keyboard and write it out to the right side. In the event that -noright is given the program simply reads the data from the left side and writes it to the screen. It the event that there is both a left and right side the program ignores the keyboard and simply reads data from the left side and writes it to the right side as well as displaying it on the screen. It is an error to have neither a left nor a right side.

Errors

Any error in a command line parameter should be caught and an appropriate error message written to the screen. At this point any error will result in the program simply terminating. If your program tries to make a connection and fails simply print an error message and terminate. We will get more elaborate about handling errors as the programs progress but for this first one just not the error you think has happens to the screen and abort.

Setting Up a Chain of piggy programs

It should be clear the we can use the program described in this specification for piggy1 to create a chain of programs on different computers that takes input from the keyboard passes it to a second program that reads the input stream and writes it to an output stream which is in turn read by a third program that reads the input stream and writes it to the screen. Furthermore, it should be noted that we can insert an arbitrary number of "middle" programs into the chain.

An example of setting up such a chain follow.

For the purposes of this example let's assume that the IP address of the computer you are logged into in the lab is 140.160.138.1. Open a terminal window and type the command

piggy1 -noright

This will create a copy of piggy that will accept a left side connection from any IP address with any source port and display what it reads from the connection to the screen.

Now use ssh to login to some computer other than the one you are sitting at, let's say its IP address is 140.160.138.2

ssh -Y your-login-id@140.160.138.2
…
On the remote machine enter the commands
piggy1 -raddr 140.160.138.1

This will create a copy of the program running on machine 140.160.138.2 that will accept a left side

connection from any machine with any source port and try to make a connection to the machine you are sitting at. Note that if the program tries to make the right side connection even before getting the left side, in this case, it will work because we have already started the "sink" program on our local computer with the first copy of piggy1.

Now open a third terminal window on the local computer and enter the commands

piggy1 -noleft -raddr 140.160.138.2

This will run a second copy of piggy on the local machine which will try to connect up to the copy we started on the remote machine (140.160.138.2). We should now have a chain of three piggies connected together. We should be able to type input into the last one we started and have that input sent to the remote machine which then send it to the local machine to the first copy of piggy we started on the local machine.

All of the port address use for the active connections (right side) defaulted to trying to connect to our reserved 367xx port. All of the passive connections, the accepting connection defaulted to listening on our default port of 367xx. Therefore, it should not matter is other people in the lab we trying to run this experiment at the same time as the ports they will be trying to connect to will be different. Note that if you do not adhere to the convention of using the port address reserved for you then things could get messed up with people hooking up to other peoples programs. There are some people from previous terms who are still working on their programs an may use port 367xx. Therefore, to prevent any possibility of colliding with them rather than using you "primary" reserved port address of 367xx please add 50 to your book number. If your book number is 25 use 36775 for your default port. The reverse side of the coin is that you should not use any ports in the 367xx range that have not been assigned to you as you may mess up other people trying to develop their programs. The router was suppose to be modifies for this term so that it will not let people remotely access the machines in cf162/4 so the only way a port conflict could happen is if it were generated from with those labs. Will we have the labs reserved for classes on Wednesdays and Fridays this means we shouldn't get any conflicts if everyone in the class obeys to conventions. If a student from a previous term happens to come into one of the labs and remotely use one of the machines you are trying to use in your testing it is possible that a port conflict will arise. This will manifest itself as an error when you try to do the bind of the port for your "left side" passive open. If you use the "add 50" trick you should be safe.

There is one more way you can get an error in trying to do the bind, you can conflict with yourself. Suppose you ran a copy of your program that successfully bound to port 36775 than your program coughed up a fur-ball and died. From the operating systems point of view the resource your process was given, the TCP port number, is still in use. The OS will notice that this resource can be reclaimed and is not actually held by any process. This doesn't happen instantaneously. This means if you try to start your program a second time and do the bind before the OS has cleaned this up in its record keeping you will get and error since the OS thinks the resource you are after is already assigned to some other process. This is such a common scenario while we are developing code and needing to run a number of tests that we have a way to prevent the bind from giving us an error. The solution is to use the socket option `SO_REUSEADDR`. In C we would call the `setsockopt` function in the socket library as illustrated below. Note that we need to set the option before we make the call to `bind` to avoid the error. There is a little confusion about this as the option at one point had two names `SO_REUSEADDR` and `SO_REUSEPORT`. You may find the `REUSEPORT` name in older examples you may encounter. While most implementations will take either name you should use the `REUSEADDR`

version as it is the accepted name in current use. This example also illustrates the use of the `QLEN` option to the listen call to set the length of the queue for how many connections that have been requested that we haven't gotten around to processing accepts for yet. All of this of course requires you to include the `socket.h` header file.

```
listensocket = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
if (listensocket < 0) {
            perror ("socket");
            exit(1);
      }

/* Eliminate "Address already in use" error message. */
int flag = 1;
if (setsockopt(listensocket,SOL_SOCKET,SO_REUSEADDR,&flag,sizeof(int)) == -1) {
            perror("setsockopt");
            exit(1);
      }

/* Bind a local address to the socket */
if (bind(listensocket, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
            perror ("bind");
            exit(1);
      }

/* Specify size of request queue */
if (listen(listensocket, QLEN) < 0) {
            perror ("listen");
            exit(1);
      }
```

If you are writing in a language other than C there will be an analogous set of calls in your language's implementation of the socket library. For your convenience I have included here at the bottom of this document copies of the man pages from section 2 of the Linux programmers manual for the functions: socket, bind, listen, accept, and connect.

You will observe in the experiment above we were very careful to start the piggy1 programs in reverse order. This is, we started with the "tail" of the chain, then started the middle, then started the "head." This was done so that when an attempt was made to open the "right side" TCP connect the program that is going to "listen" for this connection will already be in place. If we had not started them in this order we would have gotten an error when making the "connect" call to the right side and there would be no server to accept out connect. In later versions of the project we will add a command line parameter called `-orderlr`. This will tell piggy to complete the left connection before attempting the right side connection. You don't need to implement this in your piggy1 program unless you want to get a head start on later assignments. You should note, however, that if you don't implement this option in piggy1 you will need to start the parts of the chain up from tail to head.

This program should be completed by Friday April 11th for testing in the labs after our first weekly exam. You do not need to have the `-luseport` or `-lacctport` command line parameters functional by Friday. They are included here to let you know they will eventually be required so you can start working on them if you wish. To receive full marks for the first program it should be able to successfully conduct a "chain of piggies" experiment as described in this document and correctly pass a stream of data around the loop in one direction.

Debugging Aids

Telnet

You may want to use the telnet program for the "head" of the chain. Telnet was the original program created as part of the TCP/IP suite on Unix to be used for remote logins to computers across the Internet. We no longer use it for this purpose as it simply transfers all information across the connection in "plain text." That is, if you typed in a user-id and a password they would simply be sent across the network in a form any snooper could easily read. We can still make effective use of telnet as a debugging tool to make a TCP connection and type in data to be sent across the connection. To use telnet simply enter the commands

```
telnet host port
```

You should, of course, use your "primary + 50" port for these experiments. In this fashion you can use telnet to play the role of the "head" of your chain and test your piggy1 program even before you implement the part that reads from the keyboard when `-noleft` is specified. As a side note, you can also use telnet to test your implementation of the `-lacctport` option. The telnet program has a -b option you can use to specify the local port address to attempt to bind to when creating the outgoing TCP connection. A copy of the tenet man pages has been included at the bottom of this document for your convenience.

Netcat

Another program you may want to investigate using for the "head" of your chain is netcat also known as nc. Netcat can help you debug many aspects of piggy1. Netcat can be used to open a TCP connect and send data to it. It can also be used to listen on a given TCP port and show you the data that comes in on that port. Try this experiment.

Open a terminal window and type the commands

```
nc -l 367xx
```

where 367xx is your "primary + 50" port. Now open another terminal window and enter the commands

```
nc 127.0.0.1 367xx
```

The IP address 127.0.0.1 is what is called a loopback address and essential means "this computer." The re is now a TCP connect between the two terminal window. When you enter lines of text on one window they will show up on the other. This is try in both directions. Hopefully, you can see how you can use netcat to simulate the behavior of the head and tail of your chain of piggies before you get around to implementing the `-noleft` and `-noright` options. I have attached a copy of the man pages for the `netcat` command to the bottom of this document.

A good test of piggy1 would be to connect together several "middle piggies" and use netcat for the head and the tail. Have netcat put the contents of a file into the chain and get the file back at the tail and

save it to a file. The compare the two files to make sure they are the same.

```
netcat -l homeip myport > filename1
… start up a chain of “middle” piggy1programs in tail to head
sequence --
netcat ip1 myport  < filename2
diff filename1 filename2
```

The first line starts netcat on you home computer, to listen for a connection on your port. Remember you can use ifconfig to determine your IP address. The last netcat line dumps the contents of filename2 into the chain. We use ip1 to symbolize whatever the IP address of the first “middle” piggy1 in the chain is. You should have logged into it with scp and kicked of a copy of piggy1 there. The last line compares the contents of filename1 and filename2 which should be identical. If you don't get back what you inserted at the head of the chain the one or more of the piggy1 programs in the middle didn't properly transfer all the data in the stream.

IPv4 versus IPv6

You may notice references to IPv4 versus IPv6 in the description of commands or library calls. For now simply assume everything will use IP4 by default and ignore anything that has to do with IPv6. You will eventually learn how each version of IP operates in extreme detail.

Summary

OK, that's it. Get to work on piggy1. You should name your program piggy1.c or piggy.py or whatever is appropriate for your implementation language. You should submit it to canvas in the place provided by ***start of class this Friday***. Get accustomed to the weekly rhythm for the course. Each Friday we will begin class in the lab with a short period, usually, 15 minutes, in which to complete the weekly on-line exam over the concepts of the course. Keep up with the reading assignments from “the Yellow Book” in order to do well on these quizzes. We will then use the remainder of the period to run tests on your piggyN program and get a grade assigned. All programming grade are given as a percentage.

As you can see my approach to teaching computer science is heavily focused of doing. After 35 years of teaching computer science I am convinced that the only way to do it is by having you implement things on the computer. The devil is in the details. The computer is the ultimate arbiter of whether you can demonstrate an understanding of a concept. You should expect to need to spend some time each day of the week writing some code on the programming assignments for this course. You should also budget time each day to devote to reading the chapters as they are covered. The weekly Friday tests will let you know if you have been keeping up with the reading from the text.

Postscript

In recent years a phenomenon has been emerging in conjunction with this course. Students that do not put the time and effort required to complete their programming assignments by the end of the term try to negotiate with me towards the end of the term to give them an incomplete so that they can have additional time to complete the projects. I have in the past fallen prey to these pleadings. Students from one term pass this information to the students in the next term and an expectation is formed that this is

an option that will be available to them. This situation in the past two terms has gotten totally out of hand. It has gotten to the point where half of the class assumes they can put off the work and negotiate with me for an extension. These students have formed an expectation of being able to get an extension and therefore fail to put in the time and effort needed to complete the project during the term. If they have a number of courses they must balance time between they devote the time to the other courses thinking they can always negotiate with me to give them more time. Many of them attempt to use this as a negotiating point with me. I have such and such I have to get completed for professor X's class and so I would like more time to complete the assignment in yours. This behavior will cease this term.

Last term I began devising projects that must be submitted in successive pieces that build on top of the last in order to force the students to keep up with their work. Even this was not sufficient to force some students to get to work on their projects. In some cases parts that were to be completed by the 5th week we still not completed by the end of finals week. Each term the negotiations start earlier and earlier in the term. This term the project I have devised will be broken down into a larger number of successive pieces each of which builds on the last. You will receive a grade for each programming assignment along the way. In that the later programs are extensions of the earlier ones you should make doubly sure that you do not get behind on these assignments early in the term as that could possibly have a cumulative effect.

I can guarantee you that when the end of the term comes you will not be able to negotiate with me to give you an incomplete and thus get more time to complete your work. If the project work you have completed at the time the course ends is not sufficient for you to pass the class, so be it, you will fail. There will be no negotiations and no extensions.

Man page for the socket functional from section 2 of the Linux Programmer's Manual

```
SOCKET(2)                  Linux Programmer's Manual                  SOCKET(2)



NAME
       socket - create an endpoint for communication

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int socket(int domain, int type, int protocol);

DESCRIPTION
       socket()  creates  an endpoint for communication and returns a descrip-
       tor.

       The domain argument specifies a communication domain; this selects  the
       protocol  family  which will be used for communication.  These families
       are  defined  in  <sys/socket.h>.   The  currently  understood  formats
       include:

       Name                Purpose                           Man page
       AF_UNIX, AF_LOCAL   Local communication               unix(7)
       AF_INET             IPv4 Internet protocols           ip(7)
       AF_INET6            IPv6 Internet protocols           ipv6(7)
       AF_IPX              IPX - Novell protocols
       AF_NETLINK          Kernel user interface device      netlink(7)
       AF_X25              ITU-T X.25 / ISO-8208 protocol    x25(7)
       AF_AX25             Amateur radio AX.25 protocol
       AF_ATMPVC           Access to raw ATM PVCs
       AF_APPLETALK        Appletalk                         ddp(7)
       AF_PACKET           Low level packet interface        packet(7)

       The  socket  has  the indicated type, which specifies the communication
       semantics.  Currently defined types are:

       SOCK_STREAM     Provides sequenced, reliable, two-way, connection-based
                       byte  streams.  An out-of-band data transmission mecha-
                       nism may be supported.

       SOCK_DGRAM      Supports datagrams (connectionless, unreliable messages
                       of a fixed maximum length).

       SOCK_SEQPACKET  Provides  a  sequenced,  reliable,  two-way connection-
                       based data transmission path  for  datagrams  of  fixed
                       maximum  length;  a  consumer  is  required  to read an
                       entire packet with each input system call.

       SOCK_RAW        Provides raw network protocol access.

       SOCK_RDM        Provides a reliable datagram layer that does not  guar-
                       antee ordering.
```

SOCK_PACKET Obsolete and should not be used in new programs; see packet(7).

Some socket types may not be implemented by all protocol families; for example, SOCK_SEQPACKET is not implemented for AF_INET.

Since Linux 2.6.27, the type argument serves a second purpose: in addition to specifying a socket type, it may include the bitwise OR of any of the following values, to modify the behavior of socket():

SOCK_NONBLOCK Set the O_NONBLOCK file status flag on the new open file description. Using this flag saves extra calls to fcntl(2) to achieve the same result.

SOCK_CLOEXEC Set the close-on-exec (FD_CLOEXEC) flag on the new file descriptor. See the description of the O_CLOEXEC flag in open(2) for reasons why this may be useful.

The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family, in which case protocol can be specified as 0. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is specific to the “communication domain” in which communication is to take place; see protocols(5). See getprotoent(3) on how to map protocol name strings to protocol numbers.

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. They do not preserve record boundaries. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a connect(2) call. Once connected, data may be transferred using read(2) and write(2) calls or some variant of the send(2) and recv(2) calls. When a session has been completed a close(2) may be performed. Out-of-band data may also be transmitted as described in send(2) and received as described in recv(2).

The communications protocols which implement a SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered to be dead. When SO_KEEPALIVE is enabled on the socket the protocol checks in a protocol-specific manner if the other end is still alive. A SIGPIPE signal is raised if a process sends or receives on a broken stream; this causes naive processes, which do not handle the signal, to exit. SOCK_SEQPACKET sockets employ the same system calls as SOCK_STREAM sockets. The only difference is that read(2) calls will return only the amount of data requested, and any data remaining in the arriving packet will be discarded. Also all message boundaries in incoming datagrams are preserved.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in sendto(2) calls. Datagrams are generally received with recvfrom(2), which returns the next datagram along with the address of its sender.

SOCK_PACKET is an obsolete socket type to receive raw packets directly from the device driver. Use packet(7) instead.

An fcntl(2) F_SETOWN operation can be used to specify a process or process group to receive a SIGURG signal when the out-of-band data arrives or SIGPIPE signal when a SOCK_STREAM connection breaks unexpectedly. This operation may also be used to set the process or process group that receives the I/O and asynchronous notification of I/O events via SIGIO. Using F_SETOWN is equivalent to an ioctl(2) call with the FIOSETOWN or SIOCSPGRP argument.

When the network signals an error condition to the protocol module (e.g., using a ICMP message for IP) the pending error flag is set for the socket. The next operation on this socket will return the error code of the pending error. For some protocols it is possible to enable a per-socket error queue to retrieve detailed information about the error; see IP_RECVERR in ip(7).

The operation of sockets is controlled by socket level options. These options are defined in <sys/socket.h>. The functions setsockopt(2) and getsockopt(2) are used to set and get options, respectively.

RETURN VALUE

On success, a file descriptor for the new socket is returned. On error, -1 is returned, and errno is set appropriately.

ERRORS

EACCES Permission to create a socket of the specified type and/or protocol is denied.

EAFNOSUPPORT
The implementation does not support the specified address family.

EINVAL Unknown protocol, or protocol family not available.

EINVAL Invalid flags in type.

EMFILE Process file table overflow.

ENFILE The system limit on the total number of open files has been reached.

ENOBUFS or ENOMEM
Insufficient memory is available. The socket cannot be created until sufficient resources are freed.

EPROTONOSUPPORT
The protocol type or the specified protocol is not supported within this domain.

Other errors may be generated by the underlying protocol modules.

CONFORMING TO

4.4BSD, POSIX.1-2001.

The SOCK_NONBLOCK and SOCK_CLOEXEC flags are Linux-specific.

socket() appeared in 4.2BSD. It is generally portable to/from non-BSD systems supporting clones of the BSD socket layer (including System V variants).

NOTES

POSIX.1-2001 does not require the inclusion of <sys/types.h>, and this header file is not required on Linux. However, some historical (BSD) implementations required this header file, and portable applications are probably wise to include it.

The manifest constants used under 4.x BSD for protocol families are PF_UNIX, PF_INET, and so on, while AF_UNIX, AF_INET, and so on are used for address families. However, already the BSD man page promises: "The protocol family generally is the same as the address family", and subsequent standards use AF_* everywhere.

EXAMPLE

An example of the use of socket() is shown in getaddrinfo(3).

SEE ALSO

accept(2), bind(2), connect(2), fcntl(2), getpeername(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), read(2), recv(2), select(2), send(2), shutdown(2), socketpair(2), write(2), getprotoent(3), ip(7), socket(7), TCP(7), udp(7), unix(7)

“An Introductory 4.3BSD Interprocess Communication Tutorial” and “BSD Interprocess Communication Tutorial”, reprinted in UNIX Programmer's Supplementary Documents Volume 1.

COLOPHON

This page is part of release 3.54 of the Linux man-pages project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.

Man page for the bind function from section 2 of the Linux Programmer's Manual

```
BIND(2)                    Linux Programmer's Manual                    BIND(2)



NAME
       bind - bind a name to a socket

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int bind(int sockfd, const struct sockaddr *addr,
                socklen_t addrlen);

DESCRIPTION
       When  a  socket  is  created  with socket(2), it exists in a name space
       (address family) but has no address assigned to it.  bind() assigns the
       address  specified  by  addr  to  the  socket  referred  to by the file
       descriptor sockfd.  addrlen  specifies  the  size,  in  bytes,  of  the
       address structure pointed to by addr.  Traditionally, this operation is
       called “assigning a name to a socket”.

       It is normally necessary to assign a local address using bind()  before
       a SOCK_STREAM socket may receive connections (see accept(2)).

       The  rules used in name binding vary between address families.  Consult
       the manual entries in Section 7 for detailed information.  For  AF_INET
       see  ip(7),  for  AF_INET6  see  ipv6(7),  for AF_UNIX see unix(7), for
       AF_APPLETALK see ddp(7), for AF_PACKET see packet(7),  for  AF_X25  see
       x25(7) and for AF_NETLINK see netlink(7).

       The  actual  structure  passed for the addr argument will depend on the
       address family.  The sockaddr structure is defined as something like:

           struct sockaddr {
               sa_family_t sa_family;
               char        sa_data[14];
           }

       The only purpose of this structure is to  cast  the  structure  pointer
       passed in addr in order to avoid compiler warnings.  See EXAMPLE below.

RETURN VALUE
       On  success,  zero is returned.  On error, -1 is returned, and errno is
       set appropriately.

ERRORS
       EACCES The address is protected, and the user is not the superuser.

       EADDRINUSE
              The given address is already in use.

       EBADF  sockfd is not a valid descriptor.
```

EINVAL The socket is already bound to an address.

ENOTSOCK
sockfd is a descriptor for a file, not a socket.

The following errors are specific to UNIX domain (AF_UNIX) sockets:

EACCES Search permission is denied on a component of the path prefix. (See also path_resolution(7).)

EADDRNOTAVAIL
A nonexistent interface was requested or the requested address was not local.

EFAULT addr points outside the user's accessible address space.

EINVAL The addrlen is wrong, or the socket was not in the AF_UNIX family.

ELOOP Too many symbolic links were encountered in resolving addr.

ENAMETOOLONG
addr is too long.

ENOENT The file does not exist.

ENOMEM Insufficient kernel memory was available.

ENOTDIR
A component of the path prefix is not a directory.

EROFS The socket inode would reside on a read-only filesystem.

CONFORMING TO

SVr4, 4.4BSD, POSIX.1-2001 (bind() first appeared in 4.2BSD).

NOTES

POSIX.1-2001 does not require the inclusion of <sys/types.h>, and this header file is not required on Linux. However, some historical (BSD) implementations required this header file, and portable applications are probably wise to include it.

The third argument of bind() is in reality an int (and this is what 4.x BSD and libc4 and libc5 have). Some POSIX confusion resulted in the present socklen_t, also used by glibc. See also accept(2).

BUGS

The transparent proxy options are not described.

EXAMPLE

An example of the use of bind() with Internet domain sockets can be found in getaddrinfo(3).

The following example shows how to bind a stream socket in the UNIX (AF_UNIX) domain, and accept connections:

```
#include <sys/socket.h>
#include <sys/un.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MY_SOCK_PATH "/somepath"
#define LISTEN_BACKLOG 50

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

int
main(int argc, char *argv[])
{
    int sfd, cfd;
    struct sockaddr_un my_addr, peer_addr;
    socklen_t peer_addr_size;

    sfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sfd == -1)
        handle_error("socket");

    memset(&my_addr, 0, sizeof(struct sockaddr_un));
                        /* Clear structure */
    my_addr.sun_family = AF_UNIX;
    strncpy(my_addr.sun_path, MY_SOCK_PATH,
            sizeof(my_addr.sun_path) - 1);

    if (bind(sfd, (struct sockaddr *) &my_addr,
            sizeof(struct sockaddr_un)) == -1)
        handle_error("bind");

    if (listen(sfd, LISTEN_BACKLOG) == -1)
        handle_error("listen");

    /* Now we can accept incoming connections one
       at a time using accept(2) */

    peer_addr_size = sizeof(struct sockaddr_un);
    cfd = accept(sfd, (struct sockaddr *) &peer_addr,
                 &peer_addr_size);
    if (cfd == -1)
        handle_error("accept");

    /* Code to deal with incoming connection(s)... */

    /* When no longer required, the socket pathname, MY_SOCK_PATH
       should be deleted using unlink(2) or remove(3) */
}
```

SEE ALSO

accept(2), connect(2), getsockname(2), listen(2), socket(2), getad–drinfo(3), getifaddrs(3), ip(7), ipv6(7), path_resolution(7), socket(7), unix(7)

COLOPHON

This page is part of release 3.54 of the Linux man-pages project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.

Man page for the listen function from section 2 of the Linux Programmer's Manual

```
LISTEN(2)                  Linux Programmer's Manual                  LISTEN(2)



NAME
       listen - listen for connections on a socket

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int listen(int sockfd, int backlog);

DESCRIPTION
       listen()  marks  the  socket referred to by sockfd as a passive socket,
       that is, as a socket that will be used to  accept  incoming  connection
       requests using accept(2).

       The  sockfd  argument  is  a file descriptor that refers to a socket of
       type SOCK_STREAM or SOCK_SEQPACKET.

       The backlog argument defines the maximum length to which the  queue  of
       pending  connections  for  sockfd  may  grow.   If a connection request
       arrives when the queue is full, the client may receive an error with an
       indication  of  ECONNREFUSED  or,  if  the underlying protocol supports
       retransmission, the request may be ignored so that a later reattempt at
       connection succeeds.

RETURN VALUE
       On  success,  zero is returned.  On error, -1 is returned, and errno is
       set appropriately.

ERRORS
       EADDRINUSE
              Another socket is already listening on the same port.

       EBADF  The argument sockfd is not a valid descriptor.

       ENOTSOCK
              The argument sockfd is not a socket.

       EOPNOTSUPP
              The socket is not of a type that supports  the  listen()  opera-
              tion.

CONFORMING TO
       4.4BSD,  POSIX.1-2001.   The  listen()  function call first appeared in
       4.2BSD.

NOTES
       To accept connections, the following steps are performed:

           1.  A socket is created with socket(2).
```

2. The socket is bound to a local address using bind(2), so that other sockets may be connect(2)ed to it.

3. A willingness to accept incoming connections and a queue limit for incoming connections are specified with listen().

4. Connections are accepted with accept(2).

POSIX.1-2001 does not require the inclusion of <sys/types.h>, and this header file is not required on Linux. However, some historical (BSD) implementations required this header file, and portable applications are probably wise to include it.

The behavior of the backlog argument on TCP sockets changed with Linux 2.2. Now it specifies the queue length for completely established sockets waiting to be accepted, instead of the number of incomplete connection requests. The maximum length of the queue for incomplete sockets can be set using /proc/sys/net/ipv4/TCP_max_syn_backlog. When syncookies are enabled there is no logical maximum length and this setting is ignored. See TCP(7) for more information.

If the backlog argument is greater than the value in /proc/sys/net/core/somaxconn, then it is silently truncated to that value; the default value in this file is 128. In kernels before 2.4.25, this limit was a hard coded value, SOMAXCONN, with the value 128.

EXAMPLE

See bind(2).

SEE ALSO

accept(2), bind(2), connect(2), socket(2), socket(7)

COLOPHON

This page is part of release 3.54 of the Linux man-pages project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.

Man Page for the accept function from section 2 of the Linux Programmer's Manual

NAME

accept, accept4 - accept a connection on a socket

SYNOPSIS

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

#define _GNU_SOURCE             /* See feature_test_macros(7) */
#include <sys/socket.h>

int accept4(int sockfd, struct sockaddr *addr,
            socklen_t *addrlen, int flags);
```

DESCRIPTION

The accept() system call is used with connection-based socket types (SOCK_STREAM, SOCK_SEQPACKET). It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. The newly created socket is not in the listening state. The original socket sockfd is unaffected by this call.

The argument sockfd is a socket that has been created with socket(2), bound to a local address with bind(2), and is listening for connections after a listen(2).

The argument addr is a pointer to a sockaddr structure. This structure is filled in with the address of the peer socket, as known to the communications layer. The exact format of the address returned addr is determined by the socket's address family (see socket(2) and the respective protocol man pages). When addr is NULL, nothing is filled in; in this case, addrlen is not used, and should also be NULL.

The addrlen argument is a value-result argument: the caller must initialize it to contain the size (in bytes) of the structure pointed to by addr; on return it will contain the actual size of the peer address.

The returned address is truncated if the buffer provided is too small; in this case, addrlen will return a value greater than was supplied to the call.

If no pending connections are present on the queue, and the socket is not marked as nonblocking, accept() blocks the caller until a connection is present. If the socket is marked nonblocking and no pending connections are present on the queue, accept() fails with the error EAGAIN or EWOULDBLOCK.

In order to be notified of incoming connections on a socket, you can

use select(2) or poll(2). A readable event will be delivered when a new connection is attempted and you may then call accept() to get a socket for that connection. Alternatively, you can set the socket to deliver SIGIO when activity occurs on a socket; see socket(7) for details.

For certain protocols which require an explicit confirmation, such as DECNet, accept() can be thought of as merely dequeuing the next connection request and not implying confirmation. Confirmation can be implied by a normal read or write on the new file descriptor, and rejection can be implied by closing the new socket. Currently only DECNet has these semantics on Linux.

If flags is 0, then accept4() is the same as accept(). The following values can be bitwise ORed in flags to obtain different behavior:

SOCK_NONBLOCK Set the O_NONBLOCK file status flag on the new open file description. Using this flag saves extra calls to fcntl(2) to achieve the same result.

SOCK_CLOEXEC Set the close-on-exec (FD_CLOEXEC) flag on the new file descriptor. See the description of the O_CLOEXEC flag in open(2) for reasons why this may be useful.

RETURN VALUE

On success, these system calls return a nonnegative integer that is a descriptor for the accepted socket. On error, -1 is returned, and errno is set appropriately.

Error handling

Linux accept() (and accept4()) passes already-pending network errors on the new socket as an error code from accept(). This behavior differs from other BSD socket implementations. For reliable operation the application should detect the network errors defined for the protocol after accept() and treat them like EAGAIN by retrying. In the case of TCP/IP, these are ENETDOWN, EPROTO, ENOPROTOOPT, EHOSTDOWN, ENONET, EHOSTUNREACH, EOPNOTSUPP, and ENETUNREACH.

ERRORS

EAGAIN or EWOULDBLOCK
The socket is marked nonblocking and no connections are present to be accepted. POSIX.1-2001 allows either error to be returned for this case, and does not require these constants to have the same value, so a portable application should check for both possibilities.

EBADF The descriptor is invalid.

ECONNABORTED
A connection has been aborted.

EFAULT The addr argument is not in a writable part of the user address space.

EINTR The system call was interrupted by a signal that was caught before a valid connection arrived; see signal(7).

EINVAL Socket is not listening for connections, or addrlen is invalid (e.g., is negative).

EINVAL (accept4()) invalid value in flags.

EMFILE The per-process limit of open file descriptors has been reached.

ENFILE The system limit on the total number of open files has been reached.

ENOBUFS, ENOMEM
Not enough free memory. This often means that the memory allocation is limited by the socket buffer limits, not by the system memory.

ENOTSOCK
The descriptor references a file, not a socket.

EOPNOTSUPP
The referenced socket is not of type SOCK_STREAM.

EPROTO Protocol error.

In addition, Linux accept() may fail if:

EPERM Firewall rules forbid connection.

In addition, network errors for the new socket and as defined for the protocol may be returned. Various Linux kernels can return other errors such as ENOSR, ESOCKTNOSUPPORT, EPROTONOSUPPORT, ETIMEDOUT. The value ERESTARTSYS may be seen during a trace.

VERSIONS

The accept4() system call is available starting with Linux 2.6.28; support in glibc is available starting with version 2.10.

CONFORMING TO

accept(): POSIX.1-2001, SVr4, 4.4BSD, (accept() first appeared in 4.2BSD).

accept4() is a nonstandard Linux extension.

On Linux, the new socket returned by accept() does not inherit file status flags such as O_NONBLOCK and O_ASYNC from the listening socket. This behavior differs from the canonical BSD sockets implementation. Portable programs should not rely on inheritance or noninheritance of file status flags and always explicitly set all required flags on the socket returned from accept().

NOTES

POSIX.1-2001 does not require the inclusion of <sys/types.h>, and this header file is not required on Linux. However, some historical (BSD) implementations required this header file, and portable applications are probably wise to include it.

There may not always be a connection waiting after a SIGIO is delivered or select(2) or poll(2) return a readability event because the connection might have been removed by an asynchronous network error or another thread before accept() is called. If this happens then the call will block waiting for the next connection to arrive. To ensure that accept() never blocks, the passed socket sockfd needs to have the O_NONBLOCK flag set (see socket(7)).

The socklen_t type

The third argument of accept() was originally declared as an int * (and is that under libc4 and libc5 and on many other systems like 4.x BSD, SunOS 4, SGI); a POSIX.1g draft standard wanted to change it into a size_t *, and that is what it is for SunOS 5. Later POSIX drafts have socklen_t *, and so do the Single UNIX Specification and glibc2. Quoting Linus Torvalds:

"_Any_ sane library _must_ have "socklen_t" be the same size as int. Anything else breaks any BSD socket layer stuff. POSIX initially did make it a size_t, and I (and hopefully others, but obviously not too many) complained to them very loudly indeed. Making it a size_t is completely broken, exactly because size_t very seldom is the same size as "int" on 64-bit architectures, for example. And it has to be the same size as "int" because that's what the BSD socket interface is. Anyway, the POSIX people eventually got a clue, and created "socklen_t". They shouldn't have touched it in the first place, but once they did they felt it had to have a named type for some unfathomable reason (probably somebody didn't like losing face over having done the original stupid thing, so they silently just renamed their blunder)."

EXAMPLE

See bind(2).

SEE ALSO

bind(2), connect(2), listen(2), select(2), socket(2), socket(7)

COLOPHON

This page is part of release 3.54 of the Linux man-pages project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.

Man Page for the connect function from section 2 of the Linux Programmer's Manual

```
CONNECT(2)                Linux Programmer's Manual                CONNECT(2)



NAME
       connect - initiate a connection on a socket

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int connect(int sockfd, const struct sockaddr *addr,
                   socklen_t addrlen);

DESCRIPTION
       The  connect()  system call connects the socket referred to by the file
       descriptor sockfd to the address specified by addr.  The addrlen  argu‐
       ment  specifies the size of addr.  The format of the address in addr is
       determined by the address space of the socket sockfd; see socket(2) for
       further details.

       If  the socket sockfd is of type SOCK_DGRAM then addr is the address to
       which datagrams are sent by default, and the only  address  from  which
       datagrams  are  received.   If  the  socket  is  of type SOCK_STREAM or
       SOCK_SEQPACKET, this call attempts to make a connection to  the  socket
       that is bound to the address specified by addr.

       Generally, connection-based protocol sockets may successfully connect()
       only once; connectionless protocol sockets may use  connect()  multiple
       times to change their association.  Connectionless sockets may dissolve
       the association by connecting to an address with the  sa_family  member
       of sockaddr set to AF_UNSPEC (supported on Linux since kernel 2.2).

RETURN VALUE
       If  the connection or binding succeeds, zero is returned.  On error, -1
       is returned, and errno is set appropriately.

ERRORS
       The following are general socket  errors  only.   There  may  be  other
       domain-specific error codes.

       EACCES For UNIX domain sockets, which are identified by pathname: Write
              permission is denied on the socket file, or search permission is
              denied for one of the directories in the path prefix.  (See also
              path_resolution(7).)

       EACCES, EPERM
              The user tried to connect to a broadcast address without  having
              the  socket  broadcast  flag  enabled  or the connection request
              failed because of a local firewall rule.

       EADDRINUSE
              Local address is already in use.
```

EAFNOSUPPORT
The passed address didn't have the correct address family in its sa_family field.

EAGAIN No more free local ports or insufficient entries in the routing cache. For AF_INET see the description of /proc/sys/net/ipv4/ip_local_port_range ip(7) for information on how to increase the number of local ports.

EALREADY
The socket is nonblocking and a previous connection attempt has not yet been completed.

EBADF The file descriptor is not a valid index in the descriptor table.

ECONNREFUSED
No-one listening on the remote address.

EFAULT The socket structure address is outside the user's address space.

EINPROGRESS
The socket is nonblocking and the connection cannot be completed immediately. It is possible to select(2) or poll(2) for completion by selecting the socket for writing. After select(2) indicates writability, use getsockopt(2) to read the SO_ERROR option at level SOL_SOCKET to determine whether connect() completed successfully (SO_ERROR is zero) or unsuccessfully (SO_ERROR is one of the usual error codes listed here, explaining the reason for the failure).

EINTR The system call was interrupted by a signal that was caught; see signal(7).

EISCONN
The socket is already connected.

ENETUNREACH
Network is unreachable.

ENOTSOCK
The file descriptor is not associated with a socket.

ETIMEDOUT
Timeout while attempting connection. The server may be too busy to accept new connections. Note that for IP sockets the timeout may be very long when syncookies are enabled on the server.

CONFORMING TO
SVr4, 4.4BSD, (the connect() function first appeared in 4.2BSD), POSIX.1-2001.

NOTES
POSIX.1-2001 does not require the inclusion of <sys/types.h>, and this header file is not required on Linux. However, some historical (BSD)

implementations required this header file, and portable applications are probably wise to include it.

The third argument of connect() is in reality an int (and this is what 4.x BSD and libc4 and libc5 have). Some POSIX confusion resulted in the present socklen_t, also used by glibc. See also accept(2).

EXAMPLE

An example of the use of connect() is shown in getaddrinfo(3).

SEE ALSO

accept(2), bind(2), getsockname(2), listen(2), socket(2), path_resolution(7)

COLOPHON

This page is part of release 3.54 of the Linux man-pages project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.

NAME

telnet — user interface to the TELNET protocol

SYNOPSIS

telnet [-468ELadr] [-S tos] [-b address] [-e escapechar] [-l user]
[-n tracefile] [host [port]]

DESCRIPTION

The telnet command is used for interactive communication with another host using the TELNET protocol. It begins in command mode, where it prints a telnet prompt ("telnet> "). If telnet is invoked with a host argument, it performs an open command implicitly; see the description below.

Options:

-4 Force IPv4 address resolution.

-6 Force IPv6 address resolution.

-8 Request 8-bit operation. This causes an attempt to negotiate the TELNET BINARY option for both input and output. By default telnet is not 8-bit clean.

-E Disables the escape character functionality; that is, sets the escape character to ``no character''.

-L Specifies an 8-bit data path on output. This causes the TELNET BINARY option to be negotiated on just output.

-a Attempt automatic login. Currently, this sends the user name via the USER variable of the ENVIRON option if supported by the remote system. The username is retrieved via getlogin(3).

-b address
Use bind(2) on the local socket to bind it to a specific local address.

-d Sets the initial value of the debug toggle to TRUE.

-r Emulate rlogin(1). In this mode, the default escape character is a tilde. Also, the interpretation of the escape character is changed: an escape character followed by a dot causes telnet to disconnect from the remote host. A ^Z instead of a dot suspends telnet, and a ^] (the default telnet escape character) generates a normal telnet prompt. These codes are accepted only at the beginning of a line.

-S tos Sets the IP type-of-service (TOS) option for the telnet connec‐ tion to the value tos.

-e escapechar

Sets the escape character to escapechar. If no character is supplied, no escape character will be used. Entering the escape character while connected causes telnet to drop to command mode.

-l user
Specify user as the user to log in as on the remote system. This is accomplished by sending the specified name as the USER environment variable, so it requires that the remote system support the TELNET ENVIRON option. This option implies the -a option, and may also be used with the open command.

-n tracefile
Opens tracefile for recording trace information. See the set tracefile command below.

host Specifies a host to contact over the network.

port Specifies a port number or service name to contact. If not specified, the telnet port (23) is used.

Protocol:

Once a connection has been opened, telnet will attempt to enable the TELNET LINEMODE option. If this fails, then telnet will revert to one of two input modes: either “character at a time” or “old line by line” depending on what the remote system supports.

When LINEMODE is enabled, character processing is done on the local system, under the control of the remote system. When input editing or character echoing is to be disabled, the remote system will relay that information. The remote system will also relay changes to any special characters that happen on the remote system, so that they can take effect on the local system.

In “character at a time” mode, most text typed is immediately sent to the remote host for processing.

In “old line by line” mode, all text is echoed locally, and (normally) only completed lines are sent to the remote host. The “local echo character” (initially “^E”) may be used to turn off and on the local echo (this would mostly be used to enter passwords without the password being echoed).

If the LINEMODE option is enabled, or if the localchars toggle is TRUE (the default for “old line by line“; see below), the user's quit, intr, and flush characters are trapped locally, and sent as TELNET protocol sequences to the remote side. If LINEMODE has ever been enabled, then the user's susp and eof are also sent as TELNET protocol sequences, and quit is sent as a TELNET ABORT instead of BREAK There are options (see toggle autoflush and toggle autosynch below) which cause this action to flush subsequent output to the terminal (until the remote host acknowledges the TELNET sequence) and flush previous terminal input (in the case of quit and intr).

Commands:

The following telnet commands are available. Unique prefixes are understood as abbreviations.

auth argument ...
: The auth command controls the TELNET AUTHENTICATE protocol option. If telnet was compiled without authentication, the auth command will not be supported. Valid arguments are as follows:

  disable type
  : Disable the specified type of authentication. To obtain a list of available types, use the auth disable ? command.

  enable type
  : Enable the specified type of authentication. To obtain a list of available types, use the auth enable ? command.

  status
  : List the current status of the various types of authentication.

  Note that the current version of telnet does not support authentication.

close
: Close the connection to the remote host, if any, and return to command mode.

display argument ...
: Display all, or some, of the set and toggle values (see below).

encrypt argument ...
: The encrypt command controls the TELNET ENCRYPT protocol option. If telnet was compiled without encryption, the encrypt command will not be supported.

  Valid arguments are as follows:

  disable type [input|output]
  : Disable the specified type of encryption. If you do not specify input or output, encryption of both is disabled. To obtain a list of available types, use ``encrypt disable ?''.

  enable type [input|output]
  : Enable the specified type of encryption. If you do not specify input or output, encryption of both is enabled. To obtain a list of available types, use ``encrypt enable ?''.

  input
  : This is the same as ``encrypt start input''.

  -input
  : This is the same as ``encrypt stop input''.

  output
  : This is the same as ``encrypt start output''.

  -output
  : This is the same as ``encrypt stop output''.

start [input|output]
Attempt to begin encrypting. If you do not specify input or output, encryption of both input and output is started.

status
Display the current status of the encryption module.

stop [input|output]
Stop encrypting. If you do not specify input or output, encryption of both is stopped.

type type
Sets the default type of encryption to be used with later ``encrypt start'' or ``encrypt stop'' commands.

Note that the current version of telnet does not support encryption.

environ arguments...
The environ command is used to propagate environment variables across the telnet link using the TELNET ENVIRON protocol option. All variables exported from the shell are defined, but only the DISPLAY and PRINTER variables are marked to be sent by default. The USER variable is marked to be sent if the -a or -l command-line options were used.

Valid arguments for the environ command are:

define variable value
Define the variable variable to have a value of value. Any variables defined by this command are automatically marked for propagation (``exported''). The value may be enclosed in single or double quotes so that tabs and spaces may be included.

undefine variable
Remove any existing definition of variable.

export variable
Mark the specified variable for propagation to the remote host.

unexport variable
Do not mark the specified variable for propagation to the remote host. The remote host may still ask explicitly for variables that are not exported.

list
List the current set of environment variables. Those marked with a * will be propagated to the remote host. The remote host may still ask explicitly for the rest.

?
Prints out help information for the environ com-

mand.

logout Send the TELNET LOGOUT protocol option to the remote host. This command is similar to a close command. If the remote host does not support the LOGOUT option, nothing happens. But if it does, this command should cause it to close the connection. If the remote side also supports the concept of suspending a user's session for later reattachment, the logout command indicates that the session should be terminated immediately.

mode type Type is one of several options, depending on the state of the session. Telnet asks the remote host to go into the requested mode. If the remote host says it can, that mode takes effect.

character Disable the TELNET LINEMODE option, or, if the remote side does not understand the LINEMODE option, then enter “character at a time“ mode.

line Enable the TELNET LINEMODE option, or, if the remote side does not understand the LINEMODE option, then attempt to enter “old-line-by-line“ mode.

isig (-isig) Attempt to enable (disable) the TRAPSIG mode of the LINEMODE option. This requires that the LINEMODE option be enabled.

edit (-edit) Attempt to enable (disable) the EDIT mode of the LINEMODE option. This requires that the LINEMODE option be enabled.

softtabs (-softtabs)
Attempt to enable (disable) the SOFT_TAB mode of the LINEMODE option. This requires that the LINEMODE option be enabled.

litecho (-litecho)
Attempt to enable (disable) the LIT_ECHO mode of the LINEMODE option. This requires that the LINEMODE option be enabled.

? Prints out help information for the mode com‐ mand.

open host [[-l] user][- port]
Open a connection to the named host. If no port number is specified, telnet will attempt to contact a telnet daemon at the standard port (23). The host specification may be a host name or IP address. The -l option may be used to specify a user name to be passed to the remote system, like the -l com‐ mand-line option.

When connecting to ports other than the telnet port, telnet does not attempt telnet protocol negotiations. This makes it possible to connect to services that do not support the telnet protocol without making a mess. Protocol negotiation can be

forced by placing a dash before the port number.

After establishing a connection, any commands associated with the remote host in /etc/telnetrc and the user's .telnetrc file are executed, in that order.

The format of the telnetrc files is as follows: Lines beginning with a #, and blank lines, are ignored. The rest of the file should consist of hostnames and sequences of telnet commands to use with that host. Commands should be one per line, indented by whitespace; lines beginning without whitespace are interpreted as hostnames. Lines beginning with the special hostname 'DEFAULT' will apply to all hosts. Hostnames including 'DEFAULT' may be followed immediately by a colon and a port number or string. If a port is specified it must match exactly with what is specified on the command line. If no port was specified on the command line, then the value 'telnet' is used. Upon connecting to a particular host, the commands associated with that host are executed.

quit Close any open session and exit telnet. An end of file condition on input, when in command mode, will trigger this operation as well.

send arguments

Send one or more special telnet protocol character sequences to the remote host. The following are the codes which may be specified (more than one may be used in one command):

abort Sends the TELNET ABORT (Abort Processes) sequence.

ao Sends the TELNET AO (Abort Output) sequence, which should cause the remote system to flush all output from the remote system to the user's terminal.

ayt Sends the TELNET AYT (Are You There?) sequence, to which the remote system may or may not choose to respond.

brk Sends the TELNET BRK (Break) sequence, which may have significance to the remote system.

ec Sends the TELNET EC (Erase Character) sequence, which should cause the remote system to erase the last character entered.

el Sends the TELNET EL (Erase Line) sequence, which should cause the remote system to erase the line currently being entered.

eof Sends the TELNET EOF (End Of File) sequence.

eor Sends the TELNET EOR (End of Record) sequence.

escape Sends the current telnet escape character.

ga Sends the TELNET GA (Go Ahead) sequence, which likely has no significance to the remote system.

getstatus
If the remote side supports the TELNET STATUS command, getstatus will send the subnegotiation to request that the server send its current option status.

ip Sends the TELNET IP (Interrupt Process) sequence, which should cause the remote system to abort the currently running process.

nop Sends the TELNET NOP (No Operation) sequence.

susp Sends the TELNET SUSP (Suspend Process) sequence.

synch Sends the TELNET SYNCH sequence. This sequence causes the remote system to discard all previously typed (but not yet read) input. This sequence is sent as TCP urgent data (and may not work if the remote system is a 4.2BSD system -- if it doesn't work, a lower case “r” may be echoed on the terminal).

do cmd

dont cmd

will cmd

wont cmd
Sends the TELNET DO cmd sequence. cmd can be either a decimal number between 0 and 255, or a symbolic name for a specific TELNET command. cmd can also be either help or ? to print out help information, including a list of known symbolic names.

? Prints out help information for the send command.

set argument value

unset argument value
The set command will set any one of a number of telnet variables to a specific value or to TRUE. The special value off turns off the function associated with the variable. This is equivalent to using the unset command. The unset command will disable or set to FALSE any of the specified variables. The values of variables may be interrogated with the display command. The variables which may be set or unset, but not toggled, are listed here. In addition, any of the variables for the toggle command may be explicitly set or unset.

ayt If telnet is in localchars mode, or LINEMODE is enabled, and the status character is typed, a TELNET AYT sequence is sent to the remote host. The initial value for the "Are You There" character is the terminal's status character.

echo This is the value (initially “^E”) which, when in “line by line” mode, toggles between doing local echoing of entered characters (for normal processing), and suppressing echoing of entered characters (for entering, say, a password).

eof If telnet is operating in LINEMODE or “old line by line” mode, entering this character as the first character on a line will cause this character to be sent to the remote system. The initial value of the eof character is taken to be the terminal's eof character.

erase If telnet is in localchars mode (see toggle localchars below), and if telnet is operating in “character at a time” mode, then when this character is typed, a TELNET EC sequence (see send ec above) is sent to the remote system. The initial value for the erase character is taken to be the terminal's erase character.

escape This is the telnet escape character (initially “^[”) which causes entry into telnet command mode (when connected to a remote system).

flushoutput
If telnet is in localchars mode (see toggle localchars below) and the flushoutput character is typed, a TELNET AO sequence (see send ao above) is sent to the remote host. The initial value for the flush character is taken to be the terminal's flush character.

forw1

forw2 If TELNET is operating in LINEMODE, these are the characters that, when typed, cause partial lines to be forwarded to the remote system. The initial value for the forwarding characters are taken from the terminal's eol and eol2 characters.

interrupt
If telnet is in localchars mode (see toggle localchars below) and the interrupt character is typed, a TELNET IP sequence (see send ip above) is sent to the remote host. The initial value for the interrupt character is taken to be the terminal's intr character.

kill If telnet is in localchars mode (see toggle localchars below), and if telnet is operating in “character at a time” mode, then when this character is typed, a TELNET EL sequence (see send el above) is sent to the remote system. The initial value for the kill character is taken to be the terminal's kill character.

lnext If telnet is operating in LINEMODE or “old line by line“ mode, then this character is taken to be the terminal's lnext character. The initial value for the

lnext character is taken to be the terminal's lnext character.

quit If telnet is in localchars mode (see toggle localchars below) and the quit character is typed, a TELNET BRK sequence (see send brk above) is sent to the remote host. The initial value for the quit character is taken to be the terminal's quit character.

reprint
If telnet is operating in LINEMODE or "old line by line" mode, then this character is taken to be the terminal's reprint character. The initial value for the reprint character is taken to be the terminal's reprint character.

rlogin This is the rlogin mode escape character. Setting it enables rlogin mode, as with the r command-line option (q.v.)

start If the TELNET TOGGLE-FLOW-CONTROL option has been enabled, then this character is taken to be the terminal's start character. The initial value for the kill character is taken to be the terminal's start character.

stop If the TELNET TOGGLE-FLOW-CONTROL option has been enabled, then this character is taken to be the terminal's stop character. The initial value for the kill character is taken to be the terminal's stop character.

susp If telnet is in localchars mode, or LINEMODE is enabled, and the suspend character is typed, a TELNET SUSP sequence (see send susp above) is sent to the remote host. The initial value for the suspend character is taken to be the terminal's suspend character.

tracefile
This is the file to which the output, caused by netdata or option tracing being TRUE, will be written. If it is set to "-", then tracing information will be written to standard output (the default).

worderase
If telnet is operating in LINEMODE or "old line by line" mode, then this character is taken to be the terminal's worderase character. The initial value for the worderase character is taken to be the terminal's worderase character.

? Displays the legal set (unset) commands.

slc state The slc command (Set Local Characters) is used to set or change the state of the the special characters when the TELNET LINEMODE option has been enabled. Special characters are

characters that get mapped to TELNET commands sequences (like ip or quit) or line editing characters (like erase and kill). By default, the local special characters are exported.

check Verify the current settings for the current special characters. The remote side is requested to send all the current special character settings, and if there are any discrepancies with the local side, the local side will switch to the remote value.

export Switch to the local defaults for the special characters. The local default characters are those of the local terminal at the time when telnet was started.

import Switch to the remote defaults for the special characters. The remote default characters are those of the remote system at the time when the TELNET connection was established.

? Prints out help information for the slc command.

status Show the current status of telnet. This includes the name of the remote host, if any, as well as the current mode.

toggle arguments ...
Toggle (between TRUE and FALSE) various flags that control how telnet responds to events. These flags may be set explicitly to TRUE or FALSE using the set and unset commands. More than one flag may be toggled at once. The state of these flags may be examined with the display command. Valid flags are:

authdebug Turns on debugging for the authentication code. This flag only exists if authentication support is enabled.

autoflush If autoflush and localchars are both TRUE, then when the ao, or quit characters are recognized (and transformed into TELNET sequences; see set above for details), telnet refuses to display any data on the user's terminal until the remote system acknowledges (via a TELNET TIMING MARK option) that it has processed those TELNET sequences. The initial value for this toggle is TRUE if the terminal user had not done an "stty noflsh", otherwise FALSE (see stty(1)).

autodecrypt When the TELNET ENCRYPT option is negotiated, by default the actual encryption (decryption) of the data stream does not start automatically. The autoencrypt (autodecrypt) command states that encryption of the output (input) stream should be enabled as soon as possible.

Note that this flag exists only if encryption

support is enabled.

autologin — If the remote side supports the TELNET AUTHENTICATION option, telnet attempts to use it to perform automatic authentication. If the TELNET AUTHENTICATION option is not supported, the user's login name is propagated using the TELNET ENVIRON option. Setting this flag is the same as specifying the a option to the open command or on the command line.

autosynch — If autosynch and localchars are both TRUE, then when either the intr or quit characters is typed (see set above for descriptions of the intr and quit characters), the resulting telnet sequence sent is followed by the TELNET SYNCH sequence. This procedure should cause the remote system to begin throwing away all previously typed input until both of the telnet sequences have been read and acted upon. The initial value of this toggle is FALSE.

binary — Enable or disable the TELNET BINARY option on both input and output.

inbinary — Enable or disable the TELNET BINARY option on input.

outbinary — Enable or disable the TELNET BINARY option on output.

crlf — If this is TRUE, then carriage returns will be sent as <CR><LF>. If this is FALSE, then carriage returns will be send as <CR><NUL>. The initial value for this toggle is FALSE.

crmod — Toggle carriage return mode. When this mode is enabled, most carriage return characters received from the remote host will be mapped into a carriage return followed by a line feed. This mode does not affect those characters typed by the user, only those received from the remote host. This mode is not very useful unless the remote host only sends carriage return, but never line feed. The initial value for this toggle is FALSE.

debug — Toggles socket level debugging (useful only to the super user). The initial value for this toggle is FALSE.

encdebug — Turns on debugging information for the encryption code. Note that this flag only exists if encryption support is available.

localchars — If this is TRUE, then the flush, interrupt,

quit, erase, and kill characters (see set above) are recognized locally, and transformed into (hopefully) appropriate TELNET control sequences (respectively ao, ip, brk, ec, and el; see send above). The initial value for this toggle is TRUE in “old line by line” mode, and FALSE in “character at a time” mode. When the LINEMODE option is enabled, the value of localchars is ignored, and assumed to always be TRUE. If LINEMODE has ever been enabled, then quit is sent as abort, and eof and are sent as eof and susp, see send above).

netdata — Toggles the display of all network data (in hexadecimal format). The initial value for this toggle is FALSE.

options — Toggles the display of some internal telnet protocol processing (having to do with telnet options). The initial value for this toggle is FALSE.

prettydump — When the netdata toggle is enabled, if prettydump is enabled the output from the netdata command will be formatted in a more user-readable format. Spaces are put between each character in the output, and the beginning of telnet escape sequences are preceded by a '*' to aid in locating them.

skiprc — When the skiprc toggle is TRUE, telnet does not read the telnetrc files. The initial value for this toggle is FALSE.

termdata — Toggles the display of all terminal data (in hexadecimal format). The initial value for this toggle is FALSE.

verbose_encrypt — When the verbose_encrypt toggle is TRUE, TELNET prints out a message each time encryption is enabled or disabled. The initial value for this toggle is FALSE. This flag only exists if encryption support is available.

? — Displays the legal toggle commands.

z — Suspend telnet. This command only works when the user is using the csh(1).

! [command]
Execute a single command in a subshell on the local system. If command is omitted, then an interactive subshell is invoked.

? [command]

Get help. With no arguments, telnet prints a help summary. If a command is specified, telnet will print the help information for just that command.

ENVIRONMENT

Telnet uses at least the HOME, SHELL, DISPLAY, and TERM environment variables. Other environment variables may be propagated to the other side via the TELNET ENVIRON option.

FILES

/etc/telnetrc	global telnet startup values
~/.telnetrc	user customized telnet startup values

HISTORY

The Telnet command appeared in 4.2BSD.

NOTES

On some remote systems, echo has to be turned off manually when in “old line by line” mode.

In “old line by line” mode or LINEMODE the terminal's eof character is only recognized (and sent to the remote system) when it is the first character on a line.

BUGS

The source code is not comprehensible.

NAME

nc — arbitrary TCP and UDP connections and listens

SYNOPSIS

nc [-46bCDdhklnrStUuvZz] [-I length] [-i interval] [-O length] [-P proxy_username] [-p source_port] [-q seconds] [-s source] [-T toskeyword] [-V rtable] [-w timeout] [-X proxy_protocol] [-x proxy_address[:port]] [destination] [port]

DESCRIPTION

The nc (or netcat) utility is used for just about anything under the sun involving TCP, UDP, or UNIX-domain sockets. It can open TCP connections, send UDP packets, listen on arbitrary TCP and UDP ports, do port scanning, and deal with both IPv4 and IPv6. Unlike telnet(1), nc scripts nicely, and separates error messages onto standard error instead of sending them to standard output, as telnet(1) does with some.

Common uses include:

- simple TCP proxies
- shell-script based HTTP clients and servers
- network daemon testing
- a SOCKS or HTTP ProxyCommand for ssh(1)
- and much, much more

The options are as follows:

-4 Forces nc to use IPv4 addresses only.

-6 Forces nc to use IPv6 addresses only.

-b Allow broadcast.

-C Send CRLF as line-ending.

-D Enable debugging on the socket.

-d Do not attempt to read from stdin.

-h Prints out nc help.

-I length
Specifies the size of the TCP receive buffer.

-i interval
Specifies a delay time interval between lines of text sent and received. Also causes a delay time between connections to multiple ports.

-k Forces nc to stay listening for another connection after its current connection is completed. It is an error to use this option without the -l option.

-l Used to specify that nc should listen for an incoming connection rather than initiate a connection to a remote host. It is an error to use this option in conjunction with the -p, -s, or -z options. Additionally, any timeouts specified with the -w option are ignored.

-n Do not do any DNS or service lookups on any specified addresses, hostnames or ports.

-O length
Specifies the size of the TCP send buffer.

-P proxy_username
Specifies a username to present to a proxy server that requires authentication. If no username is specified then authentication will not be attempted. Proxy authentication is only supported for HTTP CONNECT proxies at present.

-p source_port
Specifies the source port nc should use, subject to privilege restrictions and availability.

-q seconds
after EOF on stdin, wait the specified number of seconds and then quit. If seconds is negative, wait forever.

-r Specifies that source and/or destination ports should be chosen randomly instead of sequentially within a range or in the order that the system assigns them.

-S Enables the RFC 2385 TCP MD5 signature option.

-s source
Specifies the IP of the interface which is used to send the packets. For UNIX-domain datagram sockets, specifies the local temporary socket file to create and use so that datagrams can be received. It is an error to use this option in conjunction with the -l option.

-T toskeyword
Change IPv4 TOS value. toskeyword may be one of critical, inetcontrol, lowcost, lowdelay, netcontrol, throughput, reliability, or one of the DiffServ Code Points: ef, af11 ... af43, cs0 ... cs7; or a number in either hex or decimal.

-t Causes nc to send RFC 854 DON'T and WON'T responses to RFC 854 DO and WILL requests. This makes it possible to use nc to script telnet sessions.

-U Specifies to use UNIX-domain sockets.

-u Use UDP instead of the default option of TCP. For UNIX-domain sockets, use a datagram socket instead of a stream socket. If a UNIX-domain socket is used, a temporary receiving socket is created in /tmp unless the -s flag is given.

-V rtable
Set the routing table to be used. The default is 0.

-v Have nc give more verbose output.

-w timeout
Connections which cannot be established or are idle timeout after timeout seconds. The -w flag has no effect on the -l option, i.e. nc will listen forever for a connection, with or without the -w flag. The default is no timeout.

-X proxy_protocol
Requests that nc should use the specified protocol when talking to the proxy server. Supported protocols are “4” (SOCKS v.4), “5” (SOCKS v.5) and “connect” (HTTPS proxy). If the protocol is not specified, SOCKS version 5 is used.

-x proxy_address[:port]
Requests that nc should connect to destination using a proxy at proxy_address and port. If port is not specified, the well-known port for the proxy protocol is used (1080 for SOCKS, 3128 for HTTPS).

-Z DCCP mode.

-z Specifies that nc should just scan for listening daemons, without sending any data to them. It is an error to use this option in conjunction with the -l option.

destination can be a numerical IP address or a symbolic hostname (unless the -n option is given). In general, a destination must be specified, unless the -l option is given (in which case the local host is used). For UNIX-domain sockets, a destination is required and is the socket path to connect to (or listen on if the -l option is given).

port can be a single integer or a range of ports. Ranges are in the form nn-mm. In general, a destination port must be specified, unless the -U option is given.

CLIENT/SERVER MODEL

It is quite simple to build a very basic client/server model using nc. On one console, start nc listening on a specific port for a connection. For example:

```
$ nc -l 1234
```

nc is now listening on port 1234 for a connection. On a second console (or a second machine), connect to the machine and port being listened on:

```
$ nc 127.0.0.1 1234
```

There should now be a connection between the ports. Anything typed at the second console will be concatenated to the first, and vice-versa. After the connection has been set up, nc does not really care which side is being used as a ‘server’ and which side is being used as a ‘client’.

The connection may be terminated using an EOF (‘^D’).

There is no -c or -e option in this netcat, but you still can execute a command after connection being established by redirecting file descriptors. Be cautious here because opening a port and let anyone connected execute arbitrary command on your site is DANGEROUS. If you really need to do this, here is an example:

On ‘server’ side:

```
$ rm -f /tmp/f; mkfifo /tmp/f
$ cat /tmp/f | /bin/sh -i 2>&1 | nc -l 127.0.0.1 1234 > /tmp/f
```

On ‘client’ side:

```
$ nc host.example.com 1234
$ (shell prompt from host.example.com)
```

By doing this, you create a fifo at /tmp/f and make nc listen at port 1234 of address 127.0.0.1 on ‘server’ side, when a ‘client’ establishes a connection successfully to that port, /bin/sh gets executed on ‘server’ side and the shell prompt is given to ‘client’ side.

When connection is terminated, nc quits as well. Use -k if you want it keep listening, but if the command quits this option won't restart it or keep nc running. Also don't forget to remove the file descriptor once you don't need it anymore:

```
$ rm -f /tmp/f
```

DATA TRANSFER

The example in the previous section can be expanded to build a basic data transfer model. Any information input into one end of the connection will be output to the other end, and input and output can be easily captured in order to emulate file transfer.

Start by using nc to listen on a specific port, with output captured into a file:

```
$ nc -l 1234 > filename.out
```

Using a second machine, connect to the listening nc process, feeding it the file which is to be transferred:

```
$ nc host.example.com 1234 < filename.in
```

After the file has been transferred, the connection will close automatically.

TALKING TO SERVERS

It is sometimes useful to talk to servers “by hand” rather than through a user interface. It can aid in troubleshooting, when it might be necessary to verify what data a server is sending in response to commands issued by the client. For example, to retrieve the home page of a web site:

```
$ printf "GET / HTTP/1.0\r\n\r\n" | nc host.example.com 80
```

Note that this also displays the headers sent by the web server. They can be filtered, using a tool such as sed(1), if necessary.

More complicated examples can be built up when the user knows the format of requests required by the server. As another example, an email may be submitted to an SMTP server using:

```
$ nc [-C] localhost 25 << EOF
HELO host.example.com
MAIL FROM:<user@host.example.com>
RCPT TO:<user2@host.example.com>
DATA
Body of email.
.
QUIT
EOF
```

PORT SCANNING

It may be useful to know which ports are open and running services on a target machine. The -z flag can be used to tell nc to report open ports, rather than initiate a connection. Usually it's useful to turn on verbose output to stderr by use this option in conjunction with -v option.

For example:

```
$ nc -zv host.example.com 20-30
Connection to host.example.com 22 port [TCP/ssh] succeeded!
Connection to host.example.com 25 port [TCP/smtp] succeeded!
```

The port range was specified to limit the search to ports 20 - 30, and is scanned by increasing order.

You can also specify a list of ports to scan, for example:

```
$ nc -zv host.example.com 80 20 22
nc: connect to host.example.com 80 (TCP) failed: Connection refused
nc: connect to host.example.com 20 (TCP) failed: Connection refused
Connection to host.example.com port [TCP/ssh] succeeded!
```

The ports are scanned by the order you given.

Alternatively, it might be useful to know which server software is running, and which versions. This information is often contained within the greeting banners. In order to retrieve these, it is necessary to first make a connection, and then break the connection when the banner has been retrieved. This can be accomplished by specifying a small timeout with the -w flag, or perhaps by issuing a "QUIT" command to the server:

```
$ echo "QUIT" | nc host.example.com 20-30
SSH-1.99-OpenSSH_3.6.1p2
Protocol mismatch.
220 host.example.com IMS SMTP Receiver Version 0.84 Ready
```

EXAMPLES

Open a TCP connection to port 42 of host.example.com, using port 31337 as the source port, with a timeout of 5 seconds:

```
$ nc -p 31337 -w 5 host.example.com 42
```

Open a UDP connection to port 53 of host.example.com:

```
$ nc -u host.example.com 53
```

Open a TCP connection to port 42 of host.example.com using 10.1.2.3 as the IP for the local end of the connection:

```
$ nc -s 10.1.2.3 host.example.com 42
```

Create and listen on a UNIX-domain stream socket:

```
$ nc -lU /var/tmp/dsocket
```

Connect to port 42 of host.example.com via an HTTP proxy at 10.2.3.4, port 8080. This example could also be used by ssh(1); see the ProxyCommand directive in ssh_config(5) for more information.

```
$ nc -x10.2.3.4:8080 -Xconnect host.example.com 42
```

The same example again, this time enabling proxy authentication with username “ruser” if the proxy requires it:

```
$ nc -x10.2.3.4:8080 -Xconnect -Pruser host.example.com 42
```

SEE ALSO

cat(1), ssh(1)

AUTHORS

Original implementation by *Hobbit* ⟨hobbit@avian.org⟩.
Rewritten with IPv6 support by Eric Jackson <ericj@monkey.org>.
Modified for Debian port by Aron Xu ⟨aron@debian.org⟩.

CAVEATS

UDP port scans using the -uz combination of flags will always report success irrespective of the target machine's state. However, in conjunction with a traffic sniffer either on the target machine or an intermediary device, the -uz combination could be useful for communications diagnostics. Note that the amount of UDP traffic generated may be limited either due to hardware resources and/or configuration settings.