# The Linux Term HOWTO

by Bill Reynolds, bill@goshawk.lanl.gov May 9, 1994

## Contents

**1 Introduction** **1**

1.1 About This Document . . . . . 1

1.2 What is term? . . . . . 1

**2 How term works** **1**

2.1 Nomenclature . . . . . 1

**3 Setting Things Up** **2**

3.1 Build term . . . . . 2

3.2 Test term . . . . . 2

3.3 Make a Transparent Link . . . . . 3

3.4 Run linecheck . . . . . 3

3.5 Try Running term . . . . . 4

3.6 Gotchas to Watch Out For . . . . . 5

**4 term clients** **5**

4.1 Other term Clients . . . . . 6

**5 X and term** **6**

**6 tredir** **7**

6.1 tredir can bite! . . . . . 8

6.2 Stupid tredir tricks . . . . . 9

6.2.1 X windows . . . . . 9

6.2.2 Mail with term . . . . . 10

**7 Automating Things** **13**

**8 Other Stuff** **15**

**9 Acks** **15**

# 1 Introduction

## 1.1 About This Document

This HOWTO attempts to clear up some of the confusion of using `term`, Michael O'Reilly's remarkable program that allows you to multiplex your serial line and set up an ersatz network connection. By and large, the documents that come with `term` are quite good, and this HOWTO is not intended to replace them. The intent of this document is to give some background on how `term` works and detail the steps in getting some of the more common networking services working under `term`. I should emphasize that this document doesn't cover everything there is to know about `term`. After reading it, you should read the `term` manpages, since they include information not contained here.

## 1.2 What is `term`?

`term` is a program, written by Michael O'Reilly (michael@iinet.com.au), that is run over a serial line to allow multiple connections to operate concurrently - *i.e.* you can be downloading a file over your modem while working on a (different) remote system via the same modem connection. `term` can also be used to open up X client windows over a serial connection. Through the `tredir` utility, `term` can provide many of the "traditional" network services: mail, news, ftp, telnet, etc. In a sense, `term` is very much like other serial protocols such as SLIP or PPP. `term` 's advantage is that it can be run entirely from user space, requiring neither kernel support nor support from system or network administrators.

# 2 How `term` works

## 2.1 Nomenclature

I'm assuming you're dialing up a system through some sort of terminal server. I use the terms "local" and "remote" to refer to the home and network connected systems respectively (unless I use them to mean something else :-).

`term` provides the local machine, which has no network connection, but is connected via a serial line to a remote machine which is in turn connected to a network, with network services . Let's look at how a machine with a "traditional" network connection provides these services. First the user invokes a program, like `ftp` or `telnet`, that requests a network service. What these program do is make a system call requesting network services. The operating system then obtains these services via it's network interface (*e.g.* it sends and receives packets over the ethernet). SLIP and PPP do exactly this, by converting your modem line into a network interface, which is in principle no different from an ethernet. The downside of this is that these protocols make the modem-connected machine part of the network, just like any other machine. This entails

all the administrative burdens associated with being a network node (more actually, since the modem link must also be administered).

In the absence of a network connection like SLIP or PPP, what does one typically do? Well, you dial up your network connected machine, read your mail, your news etc, if you need a file, you first transfer it to the remote machine and then download it to your local machine using `kermit` or its ilk. This is a bit of a pain, especially since you can only really do one thing at a time that uses your modem link. The idea behind `term` is basically to automate and multiplex this process. `term` is invoked on both the local and remote machines, and the two processes communicate with one another over the modem line. When you need a network service, you make a request to the local `term` daemon, which forwards the request to the `term` daemon on the remote, network-connected, machine. The result is then returned over the modem line.

To be more concrete, say you want to retrieve a file by `ftp`. First you need a version of `ftp` that can speak to `term`. You invoke this `termftp` as you do a regular `ftp`, say `termftp nethost.gov`, but this special version makes its network request to the local `term` daemon instead of the kernel. The local `term` forwards this request, over the modem line, to the remote `term`, which opens an ftp connection to `nethost.gov`, and transmits the data back over the modem link. `term` is smart enough to have many different things going on at once, so you can have several different network sessions using the same modem link, *e.g.* you can be logged into another distant host via `termtelnet` while the `termftp` transfer is going on.

If this is too abstract (or unclear) don't worry, the important piece of information to get out of this section is that there are *two* copies of `term` running, one on each end of the modem link.

# 3 Setting Things Up

## 3.1 Build `term`

Hopefully, this should just involve a `make`, if this doesn't work, you may have to do some hacking. Make sure that you've got the same version on all machines you want to run `term` on - different versions of `term` are incompatible. Seek out and destroy any old binaries on your system - or know how to avoid them.

## 3.2 Test `term`

Do a `make test` to build `term`'s test daemon. Test works by running two copies of `term` on your system, a "local" and a "remote" copy. Both of these will read your `termrc`, so you can adjust their behaviour. Run `test` as `./test` so as to avoid your system's `test`. You should now be able to do a `trsh` and a `tupload` (try `tupload ./term /usr/tmp` - you should get a copy of the `term` binary in `/usr/tmp`). The local `term`'s output should show up in `local.log`, the remote's in `remote.log`. You can start `term` up with a `-d255` flag to enable debugging output to be written to these files, or enable debugging in your `termrc`.

## 3.3 Make a Transparent Link

Presumably, you can establish a modem connection between your local and remote hosts. Typically, your dialing into some kind of terminal server and connecting to your remote host from there. You are also

using some kind of terminal software, such as `kermit` or `seyon` to talk to your modem (the examples in this document will use `kermit`, since that is what its author uses). If you're having trouble with your modem, or your terminal software, take a look at the Serial HOWTO, that should help you out.

Having established your link, you want to make it as transparent as possible. Check the commands on the terminal server (help or ? will usually get you started). Go for the 8 bit options whenever possible. This may mean changing the way you log in to a system, *e.g.* if the server uses rlogin, you may have to use it and give it the -8 flag to make it transparent. Especially watch out for xon/xoff flow control, you don't want that, try to enable rts/cts (hardware) flow control. You may need to check your modem docs to learn how to configure it to do 8-bit rts/cts communications.

## 3.4 Run linecheck

`linecheck` is program that is supplied that with `term` that checks the transparency of a link, providing configuration information that `term` needs to run correctly. `linecheck` will send each of the 255 possible eight bit characters over the link and verify that each was transmitted successfully. `term` needs to be configured to deal with characters that cannot be transmitted over the link, and `linecheck` determines what characters these are. You use `linecheck` after you've established as transparent a modem link as possible. To run `linecheck`, do the following

1. On the remote system, run `linecheck linecheck.log`.
2. Suspend your comm program (`^Z` under kermit) (otherwise it will steal characters from linecheck).
3. On the local system run `linecheck linecheck.log > /dev/modem < /dev/modem`.

When `linecheck` is done, you'll find a set of numbers at the bottom of the `linecheck.log` files. These should be escaped in the termrc at the other end of the link. For example, on my system, my local `linecheck.log` said nothing and my remote `linecheck.log` said to escape 29 and 157. Therefore, my local `termrc` escapes these characters and my remote `termrc` escapes nothing. If I `escape` a character on one end, I have to also `ignore` it on the other, so, in this example, I'll have to ignore 29 and 157 on my remote system.

If linecheck hangs, try using

```
linecheck linecheck.log 17 19 > /dev/modem < /dev/modem
```

this will escape your xon/xoff (flow control) characters, which will hang your line if you've got software flow control. If this solves the hanging problems, you'll want to `escape /ignore` 17/19 in both `termrc`'s. If your terminal server has other demonic characters that will shut it down, try running linecheck with those characters escaped, as above. You can spot these characters if `linecheck` hangs. If this is the case, kill it, then look in the logfiles. The last characters transmitted are likely to be the culprits. Try it again with these characters escaped.

In summary, my local termrc has the lines

```
escape 29
escape 157
```

and my remote termrc has the lines

```
ignore 29
ignore 157
```

since my *remote* `linecheck.log` said to escape 29 and 157.

## 3.5 Try Running `term`

Log into the remote system, making the link as transparent as possible (if you haven't already). Fire up `term` on the remote end. I use the following:

```
exec term -l $HOME/tlog -s 38400 -c off -w 10 -t 150.
```

Let's run down each option one by one (note that I could just as easily have put these options in my termrc. I did it this way because it saves editing a file while getting `term` set up).

`exec` means to destroy your current shell, running the given program in its place. I `exec` things because I don't intend to use my login shell again, so it's just wasting memory. If you're debugging the link and can reliably kill the remote `term,` you might not want to do an exec.

`-l $HOME/tlog.` This logs errors to the file `tlog` in my home directory. Very useful for debugging. No reason not to do this.

`-s 38400` : I've got a 14400 baud modem, with compression. For optimal compression ratios, I want to be able to push bits down the pipe as fast as possible. For a slower modem, you should use something lower. Note that if you have a slower machine with 16450 uart on your serial port, high baudrates can cause data loss by overloading the chip on your serial port. `term` will recover from this, but if you see a lot of error messages in your log file, (or get overrun warnings from linux kernel versions 0.99pl15 and up) you again might want to lower this number.

`-c off` : This turns data compression off. I've got a compressing modem, and I don't want to compress things twice.

`-w 10 -t 150` : Again, these are options to optimize my fast modem link. I have set my window to 10 and my timeout to 150. This is per the recommendation in the `term_setup` man page.

Suspend your comm program (`^Z` under kermit). You don't want it running while `term` is running, because it will fight with `term` over the serial port. If you can convince your modem to not hang up when you exit your comm program (when it toggles DTR), you could just exit the program at this point.

Now run `term` locally. I use:

```
term -r -c off -l $HOME/tlog -s 38400 -w 10 -t 150 \ < /dev/modem > /dev/modem &
```

The only option here that is different from the other end is the `-r` option. You need this on one end, otherwise `term` clients will spontaneously die on you. I need to tell `term` where the modem is, so I point both standard input and output at `/dev/modem` (that's what the `<` and `>` do). I also run it in the background, so I can use this screen for something else if I want to.

`term` should work now :-). Try a `trsh`, and see what happens. If you hang, or your link seems slow, take a look at your `tlog` on each end. Are you getting timeouts or error messages? If so, then you've configured something incorrectly. Try again (after you've finished reading this :-). Note that the connection won't seem blazingly fast, especially if you're using compression - it will be a little jumpy. The real speed comes in during file transfer and the like.

### 3.6 Gotchas to Watch Out For

Are you running the right binary? `term` has been updated quite a lot, and many systems have different versions of the programs floating around. Make sure you're using the right version. Note that this applies to linecheck too. You can use bash's `type -a` or the `whereis` command to find which program is being run. `term` versions after 1.11 should print out their version number when they start up. (Although version 1.14 claims to be 1.12. Sigh.)

Do you have the right `termrc`? Is it called `$HOME/.term/termrc`? Some systems have pre-installed termrc's, make sure they're gone before you set things up. If you're running things as root, lookout for `/.term`. `term` creates files (sockets actually) while it's running, so it has its own directory, `$HOME/.term`, where the file `termrc` goes (note, there is *no* leading dot in `termrc`!).

Once you've got `term` running, you might want to try to get things optimized. A good way to measure the speed of your link is to run `tmon` in one window while up/downloading a file in another. Try both (big) text files and compressed files, the plain text should go a factor of two-ish faster than the compressed files. The parameters you want to fiddle with are compression (`-c`), windows (`-w`) and timeout (`-t`).

`Compression` you want this on if you don't have a compressing modem. If you do have such a modem, turn compression off, otherwise you'll be compressing things twice, which typically *increases* the amount of data transmitted. Compressing modems are ones that use the MNP-5 or V42.bis protocols. Check your modem documentation and the message when your modem connects.

`Windows` this is the number of chunks of data, or packets, that `term` will let go over the line before it gets an acknowledgment (or ack) from the remote `term`. For fast modems, increasing this can be a win, for slower links this can overwhelm the remote end.

`Timeout` the time `term` will wait for an ack. If you've increased windows, and you're getting timeouts in your logfile, try increasing this.

For 14400/V42.bis, I use `-c off -w 10 -t 150`. I typically get around 1700 cps on compressed files and 3500 cps on ascii files using tupload.

## 4 term clients

`term` provides several default clients. They include `trsh`, `tmon`, `tupload`, `tredir` and `txconn`. This section

will deal with the first three. The others each have their own section. No term client will work until you have established a term link.

tmon is a simple utility to monitor the statistics of your link. It prints a time histogram of characters transmitted and received. It is invoked simply as tmon. Since around version 1.11, tmon has had a bug that causes some information to be garbled. Send me mail if you'd like a patch to fix this.

trsh is similar to rsh. Without arguments, it spawns an interactive shell on the remote system (*i.e.* it logs you in). trsh is one of the primary means of accessing the remote end of the link via term. If given an argument, trsh executes that argument as a command on the remote system. For example trsh ls would give you a listing of the files in your home directory on the remote system.

tupload is the term 's basic file transfer program. It will transfer a file, given as it's first argument, to the opposite end of the link. By default, the files will be put in the same directory that you invoked term from on the other side. To put files in another directory, give it's name as a second argument to tupload. For example, if I want to put a copy of the file term114.tar.gz in /usr/tmp on the remote system, I would type tupload term114.tar.gz /usr/tmp. There is no client to do a download - to download files, trsh to the remote system and do a tupload.

### 4.1 Other term Clients

There are now a number of programs that have been modified to work with term, they include telnet, ftp. Mosaic and many others. They can be found on sunsite.unc.edu in /pub/Linux/apps/comm/termstuff.

## 5 X and term

term allows users to open up X windows on the local machine from clients that are running on a machine on the network. This is done by using the txconn client. txconn is executed on the remote, network-connected machine, it is invoked simply as txconn. It goes into the background and returns a number on the standard output, this number is the display number that clients should use to access the X server on the local machine. An example should make this clear. I am logged in, via trsh, to my remote term host, named foo. On foo, I do the following

```
foo$ txconn
Xconn bound to screen 10
:10
foo$
```

Now, on any host that I wish to run an X client, that is to display on my local machine's X server, I do setenv DISPLAY foo:10. Now when I start the client, it will try to connect to screen 10 on machine foo, but txconn is listening to this screen, and will forward all X protocol packets via term to the X server on the local host - *i.e.* the window will open up on your local machine.

It is possible to go the other way - run a client on your local machine and have it open up a window on a remote machine on the network, however we will defer explaining this until after we have discussed tredir.

txconn is not terribly secure, anyone can connect to your local server via term and perform all sorts of mischief. If you're worried about this sort of thing, it might be a good idea to consider using xauth to authorize your connections. Consult the xauth man page.

The X protocol is not very efficient, it wastes some bandwidth. This is usually not a problem over an ethernet, but can be murder over a modem. X11R6 is supposed to introduce a low bandwidth version of the X protocol, LBX. However, this is not of much use, since R6 has not been released at the time of this writing. There is a utility named sxpc which compresses the X protocol, improving response over serial lines. sxpc includes a writeup on how to get it working with term, and is recommended. You can get sxpc from ftp.x.org in /contrib. The sxpc package also explains how to use xauth, so it is doubly recommended.

# 6 tredir

tredir is one of term's most powerful utilities, allowing most important network services to be performed over a term link. Before we explain how to use tredir, it is necessary to give some background on network services. We've talked about network services before, but we haven't said exactly what they are. Services are just that - services that are provided by the network. Examples of services include telnet, which provides logins between machines, the File Transfer Protocol, ftp, which transfers files between machines, and smtp, the Simple Mail Transfer Protocol, which is used whenever you send electronic mail. Each network service has a *port number* associated with it. The mapping of port numbers to services is given in the file /etc/services. This file should be the same on all internet-connected machines.

How are these services invoked? Each networked machine runs a daemon called inetd, which listens for attempts to connect to the network ports. These requests can come from either the network or the local machine. A network service is obtained by connecting to a particular inetd port. When a network request is made, inetd knows exactly which service is involved by the port number that request is made on. If inetd is configured to do so, it then provides the relevant service to the requesting connection. inetd's configuration is given by the file /etc/inetd.conf, which has a list of the services that inetd provides. For more information, see the man pages for inetd and inetd.conf.

You can communicate directly with network services by using telnet (*n.b.* not termtelnet). For example, to talk to the sendmail, (or smtp) daemon on machine machine_name, you can do a telnet machine_name smtp, or telnet machine_name 25, (since 25 is the number assigned to smtp in /etc/services). You should get a polite greeting from the daemon on the remote machine. This is a very useful trick for debugging network problems and checking ports redirected with tredir (see below).

tredir works very much like inetd. It runs in the background as a daemon, listening to the network ports, waiting for a request. When a request for a service is made, instead of providing that service, as inetd does, tredir forwards the request over the term link to the remote term, which makes the request over the network, returning the result back over the link to the local client. tredir can forward the request to any machine on the network, but by default sends it to the machine on the other end of the term link. tredir "redirects" network services over the term link (hence the name "tredir").

An example should make this clear. Let's redirect a local port to the telnet port on the remote machine. To do this we would do tredir 2023 23. Now, anyone who connects to port 2023 on the local machine will be redirected to port 23 (telnet) on the remote machine. Here's an example session, the local machine

is `mymachine.modem.home` and the remote machine is `netsun`.

```
$ tredir 2023 23
Redirecting 2023 to 23
$ telnet localhost 2023
Trying 127.0.0.1...
Connected to mymachine.modem.home
Escape character is '^]'.


SunOS UNIX (netsun)
login:
```

This example is actually quite useful. If I were instead to do the tredir on `netsun`, I could then telnet in to `mymachine` from the network, simply by connecting to the redirected port on the networked machine (using telnet) - *i.e.* `telnet netsun 2023`.

The general principle in using `tredir` is to redirect the desired service to a machine on the network. Our next example will allow us to read news on the local machine over our `term` link from a news server on the network. News is provided by the nntp service, port number 119. All decent newsreaders allow you to specify what port number they will use, either via a configuration file or an environment variable. Let's specify this local port to be 2119. Now, let's say that our news server is `news.domain.org`. We will redirect port 2119 to port 119 on `news.domain.org`, we will then tell our news reading software that the nntp server is located at port 2119 on the local host. Since this will depend on the news reader that you use, I will just test the link with `telnet` instead of firing up a newsreader:

```
$ tredir 2119 news.domain.org:119
Redirecting 2119 to news.domain.org:119
$ telnet localhost 2119
Trying 127.0.0.1...
Connected to mymachine.modem.home.
Escape character is '^]'.
200 news.domain.org InterNetNews NNRP server INN 1.4 07-Dec-41 ready
(posting ok).
```

If you can get this far, all you have to do is configure your news reader to be able to read news via `term`. (*n.b.,* if you read news like this, be sure that in all your posts you set a Reply-To: header to an network email address that you can be reached at, otherwise people who want to get in touch with you will be sending mail to whatever (wrong) data your newsreader puts in the From: header).

## 6.1 `tredir` can bite!

The astute reader, after reading the last example will be wondering why port 2119 was redirected to port 119 - since newsreaders default to port 119, why couldn't I do a `tredir 119 news.domain.org:119` and skip

the newsreader configuration? The answer is that all ports numbered less than 1024 are "reserved ports", and only the superuser can listen on them. If one is willing to take a security risk and make `tredir` an suid program, or run `tredir` as root, then one can redirect reserve ports and avoid the hassle of renaming services.

Another problem with using reserved ports is that `inetd` is often already listening to these ports, and only one program at time can listen to a port. In order to use such a port, you must change `inetd.conf` so that `inetd` no longer listens on the port you want to redirect. This is most easily done by commenting out the line with the offending service by putting a `#` character at the beginning of the line. The superuser must then send `inetd` a HUP signal to get it to reread its configuration.

All of the services we've been discussing so far are TCP (Transmission Control Protocol) services - TCP is just a protocol for sending packets (data) over a net. `term` only works with TCP packets. There are other network protocols, such as UDP (the User Datagram Protocol) that are used for things like network file systems, that `term` cannot handle. If you need to obtain a UDP service, like NFS, over a `term` link, you are out of luck. In this case, it would behoove you to consider PPP or SLIP.

## 6.2 Stupid `tredir` tricks

In this section we will describe some of the more common uses for `tredir.` We have already described how to redirect nntp and telnet services, here we will give some more complicated examples.

### 6.2.1 X windows

In a previous section, we described how to get an X client running on the network to open a window on your home machine using `txconn.` The same technique could be used on your home machine to display a client on the machine at the remote end of your `term` link. But how does one display an X client on a network machine that is not the remote end? The answer lies in knowing that X uses a particular network service just like the other programs we've been discussing. An X server listens for network request on a port whose number is given by the formula $port = 6000 + displaynumber$, *e.g.* an X server managing screen 0 on a machine would listen to port 6000, if it were managing screen 2, it would listen on port 6002. When you set your DISPLAY environment variable to `xmachine:n,` your X clients will try to connect to port $6000 + n$ on `xmachine.`

We can use this to trick X clients on your local machine to open up windows on remote displays. Let's say I want to open up an `xterm,` running on my local machine, on display 0 of machine `xmachine,` which is running someplace on the network. I first pick a local display number, say 2 (don't use 0, since that's what your local X server will be using). I will map this display to display 0 on `xmachine.` In terms of ports, this means I want to redirect the local port 6002 to the remote port 6000. I do the following

```
$ tredir 6002 xmachine:6000
$ setenv DISPLAY localhost:2
$ xterm
```

This should open up an `xterm` on machine `xmachine.` Note that I set the DISPLAY to `localhost:2.` This is because X clients will sometimes use unix domain sockets instead of internet domain sockets, at their own option, when connecting to a local display, if DISPLAY is set to `:2. localhost:2` says to use a tcp connection.

Note that as far as `xmachine` is concerned, the X request is coming from the machine on the remote end of your `term` link (`remotemachine`) - so if you need to authorize the connection, you should either do an `xhost + remotemachine` on `xmachine` or use `xauth` to update the `.Xauthority` file on your local machine for display number 2, using the key from `xmachine.`

Again, to speed up X connections, you can use the program `sxpc,` which includes an explanation of how to use `tredir` to establish the link and authorize it using `xauth.`

### 6.2.2 Mail with `term`

Well, you asked for it. Electronic mail has the justifiable reputation of being one of the most difficult things to get working right on a UNIX system. To really get `term` working correctly with mail means that you have to understand how mail works, which is beyond the scope of this document. To learn more about mail, you should consult a book on UNIX system administration and/or the comp.mail.misc FAQ, available for anonymous ftp on rtfm.mit.edu in `pub/usenet/comp.mail.misc.` There are also currently two packages available for anonymous ftp on `sunsite.unc.edu` that will help you get mail running under `term` - they are `term.mailerd+smail` by Byron A. Jeff and the `BCRMailHandlerXXX` by Bill C. Riemers.

That being said, we'll give a thumbnail description of how mail works. There are two parts to getting mail running, sending messages and receiving messages. We'll begin with sending messages from your local box to the network.

There are two classes of mail programs. The first is the mail user agent (MUA). MUA's help you read, compose and send messages. Examples of MUA's are `elm`, `pine`, `Mail` and `vm`. MUA's don't really do any networking, they just put the messages together - the real work of sending mail is done by the second class of mail programs, the mail transfer agents (MTA's). These are invoked by the MUA's. They take the message, decide where to send it by looking at the address, and then actually deliver it over the network.

The two most common MTA's on Linux systems are `sendmail` and `smail`. The basic idea is to get your MTA to connect to another MTA running on a machine on the net that will know what to do with your message. This is done by redirecting a local port to the smtp port on the net machine. You then have to tell you MTA to take any message it doesn't know what to do with and send it out over the redirected port on your local machine to the MTA on the remote machine, which will then route your message to its correct destination.

How do we do this using `smail`? We first redirect a port to the smtp port on the network mail machine (`mailhost`):

```
tredir XXXX mailhost:25
```

here XXXX is the port number that the `smail` on the localhost will connect to (note that I have to give this port a name in my `/etc/services` to get `smail` to recognize it). `smail` has several configuration files that usually live in `/usr/local/lib/smail.` The ones we care about are `config, routers` and `transports.`

Note that I'm assuming you've already got `smail` configured correctly for local mail - delivery to files and pipes and such. Again, consult the documentation if you don't.

In the file `config`, we put the following definition:

```
smart_path=localhost
```

localhost is the machine that that `smail` connects to when it doesn't know what to do with a message.

In `routers` we put

```
smart_host:
driver=smarthost,
transport=termsmtp;
path = localhost
```

In `transports` we put

```
termsmtp:       driver=tcpsmtp,
        inet,
        return_path,
        remove_header="From",
        append_header="From: YOUR_NET_ADDRESS",
        -received,
        -max_addrs, -max_chars;
        service=YOUR_SMTP_SERVICE,
```

In the above, the header lines change the `From` header in all your outgoing mail to the address, `YOUR_NET_ADDRESS`, which is the network address you want mail for sent. If more than one user is going to be using your `term` link, you'll have to do something more fancy, like keep a database of local user's network addresses and insert these in the `From:` headers.

The service line is the name of the local portnumber that you've redirected to the smtp port on the network connected machine. On my version of `smail`, I can't just set this to a number, I have to set it to a name, like "foo", and then define "foo" in my `/etc/services` to be the number of my redirected port. If you use an suid `tredir` and just redirect the smtp port (25), you don't need to define this.

This should be enough to get you going. If you decide to use `sendmail`, the principles are the same but the details differ. Ronald Florence (ron@mlfarm.com) tells me tells me that the stock Sun `sendmail` won't send multiple queued messages over a redirected port; BSD sendmail 8.6.9 works fine. He made the following changes to `sendmail.cf` to get it working with term. In his case, the default sendmail port (25) is used for SMTP traffic over a local ethernet, so Internet mail is forwarded to a redirected TCP port.

```
#
```

```
#Create the termsmtp mailer, which sends mail via a re-directed TCP port
#
Mtermsmtp,P=[TCP], F=mDFMuCXe, S=22, R=22, A=TCP $h PORTNUMBER
```

Here, `PORTNUMBER` is the number of the redirected port on the local machine. This should be an unused port over 2000. We next tell sendmail to which machine to connect to, and set termsmtp as the default mailer.

```
#
# major relay mailer
#
DMtermsmtp
#
# major relay host: use the $M mailer to send mail to other domains
#
DR HOSTNAME
CR HOSTNAME
```

Here `HOSTNAME` is the name of your local host (does `localhost` work?). The last entry goes under Rule 0 to forward Internet mail.

```
# Pass other valid names up the ladder to our forwarder
R$*<@$*.$+>$*           $#$M    $@$R $:$1<@$2.$3>$4     user@any.domain
```

When the term connection is established to the Internet host, run the following commands on the local machine.

```
tredir PORTNUMBER internet.host:25
/usr/lib/sendmail -q
```

We now turn to receiving electronic mail using `term.` We'll assume that mail is sent to your account on the network machine `mailhost.` The simplest solution is to just use `trsh` or `termtelnet` to log on to `mailhost` and read your mail on there. However, it is also possible to have your mail automatically downloaded to your local machine. One way to do this is to use the Post Office Protocol, (POP). POP was designed for exactly this purpose: to deliver mail to machines that have intermittent network connections. To use POP, you must have a POP server installed on `mailhost.` Assuming that you do, you can then use a POP client to download your mail every few minutes. This is done, as you might expect, using `tredir.` The POP service is 110 (*n.b.* that there is an older protocol, POP-2, which uses port 109, in this document we describe POP-3, which is the latest version of POP). There are several POP clients available. One, written using the script language `perl` is pop-perl-1.X, written by William Perry and maintained by myself - it can be found on `sunsite` in `/pub/Linux/system/Mail.`

To use POP, you redirect a local port to port 110 on `mailhost` and configure your client to retrieve your mail from localhost using the local port. As an example, we'll assume that there is a POP server running on `mailhost.` We'll redirect the local port 2110, and fire up the pop-perl client:

```
$ tredir 2110 mailhost:110
Redirecting 2110 to mailhost:110
$ pop
Username: bill
Password: <enter your password for mailhost>
Pop Host: localhost
Pop Port: 2110
Starting popmail daemon for bill
```

If you do not have a POP server available, the BCRMailHandler package has a program to download your mail over a `term` link to your local machine. I have not used it, but anyone who has is welcome to comment.

# 7 Automating Things

Now that you know how to get all of your network services over `term`, it would be nice to set things up so your link is set up and configured automatically. There are basically an infinite number of ways of doing this, depending on what comm program you use and how you log in to your remote system.

One program, that I haven't used, but I've heard is quite nice, is `fet`: a front end for `term`. It is designed to log you into a remote system and fire up `term` and all your `tredir`'s. Any comments on `fet` would be most welcome.

I'll now give an example of a set of commands that use `kermit` to log into the remote system and then performs all of the `term` initializations. Obviously, if you use these examples, you will have to modify them for your own login procedures.

The command which is actually invoked is the shell script `knet`, given by:

```
#!/bin/sh
/usr/bin/kermit -y $HOME/.kerm_term > $HOME/klog < /dev/null 2>& 1
exec $HOME/bin/tstart >> $HOME/klog 2>& 1
```

The script `.kerm_term` is given by:

```
pause 2
# The number you want to dial
output atdtXXXXXXX \13
# Login to the terminal server
input 145 {name: }
output MYNAME \13
input 3 {word: }
output MYPASSWORD \13
```

```
input 5 {xyplex>}
# Make the line transparent
output term telnet-t \13
output term stopb 1 \13
# Connect to the remote host
output telnet remotehost.somedomain.org \13
input 10 {ogin: }
output MYOTHERNAME \13
input 3 word:
output MYOTHERPASSWORD \13
pause 5
# Fire up term on the remote host
output exec term -s 38400 -l $HOME/tlog -w 10 -t 150  \13
# Suspend kermit
!killall -STOP kermit
```

and finally, the script `tstart` which fires up the local `term` and its clients is given by

```
#!/bin/sh
#
# Fire up term
#
/usr/local/bin/term -l $HOME/tlog -s 38400 -r -w 10 -t 150 < /dev/modem > /dev/modem &
#
# Let it get going
#
sleep 3
#
# This lets mail get out, can read news here, can pick up my mail here
#
/usr/local/bin/tredir 2025 25 2119 newshost:119 2110 pophost:110
#
# So I can open up Xwindows here
#
/usr/local/bin/trsh -s txconn
#
# So I will receive mail....
#
/usr/local/bin/pop
#
# Clean out the queue, in case of boo-boos
#
/usr/bin/runq
```

```
#
# Done now
#
echo ^G^G > /dev/console
```

As I said, there are zillions of ways to do this, these are just meant as examples to get you started.

# 8 Other Stuff

I would like future versions of this HOWTO to include some more stuff, however I've run out of gas for the time being (maybe next month). Any help would be greatly appreciated. Some things that might be nice include,

- A complete list of `term` clients.
- How to port a client to `term`
- Other versions of `term` : Olaf Titz's patches, the `term+share` package. Any others?
- Suggestions.

Anyways, if you have comments, criticisms, suggestions, or anything else to say about this document, please fire away. I'm Bill Reynolds and can (currently) be reached at `bill@goshawk.lanl.gov`.

# 9 Acks

Thanks go to a lot of people. First and foremost to Michael O'Reilly and all the developers of `term`, who have provided us with such a great tool. I would also like to thanks everyone who gave feedback and contributed to this HOWTO, they include Ronald Florence, Tom Payerle, Bill C. Riemers, Hugh Secker-Walker, Matt Welsh and everybody whose mail I've accidently deleted or forgotten to mention.