# TransSys PNI User Manual

Version 1.13

**Louis A. Mamakos, TransSys, Inc.**

$Header: /local/SRCS/slip2/doc/pni.texinfo,v 1.24 1994/10/03 03:49:34 louie Exp $

# 1 Introduction

## 1.1 About TransSys PNI

TransSys PNI (Pseudo Network Interface) software is used to create virtual network interfaces on your NEXTSTEP system which don't directly correspond to hardware network interfaces. By creating pseudo- or virtual-network interfaces, network connectivity can be extended over different types of media using a number of different mechanisms.

### 1.1.1 PNI as SLIP

The TransSys PNI software can be used to build wide-area networks by utilizing serial connections as network links. This is done by using the SLIP encapsulation protocol over the links to move network packets between systems using the serial ports as the network interface, rather than the Ethernet interface. Once a network connection is established between your NeXT computer and the remote network access server or router, it is possible to use the connection to support multiple simultaneous connections for remote login, file transfer, mail, or and other TCP/IP based network protocol.

Of course, a 9.6 kb/s serial link isn't nearly as fast as a 10 Mb/s Ethernet port. You'll probably not want to try to use bandwidth intensive applications, like *NFS* (the Network File System).

This software is usually used by a end-system which dials into some network access server which also expects to provide network connections over serial links. This software contains a kernel or operating system level driver that transports IP datagrams across the serial link using a trivial encapsulation called *SLIP* (Serial Line IP). You have to be talking to another device which also knows how to do SLIP, and not just a terminal server.

### 1.1.2 PNI as CSLIP

This software is also capable of implementing Van Jacobson-style TCP Header compression, which is a technique to vastly improve interactive response time for TCP connection across the SLIP link by compressing or eliminating redundant header information. This is commonly referred to as CSLIP.

### 1.1.3 PNI as virtual network link

It is possible to create *tunnels*, or virtual network connections, over existing networks to layer a new logical network (where the traffic is optionally encrypted for privacy purposes) over some existing IP infrastructure. In this instance, rather than using serial lines to provide transport for the packets being sent on a network interface, existing network connections are used to carry traffic for the virtual link.

Once the link has been brought up, you can arrange for the link to be dropped When the link has gone idle for some configured period.

Extensive filtering is available which can be used to implement site-specific security policy. You can restrict the flow of traffic both arriving on a network connection as well as the traffic being sent on a connection. These filters can be updated dynamically and programmatically.

### 1.1.4 Is PNI for you?

TransSys PNI is very much more UNIX tool-like, rather than a traditional NEXTSTEP application. Partly, this is because its function is to more support the entire *system*, rather than a particular user. It is also useful when there are no users active on the NEXTSTEP system at all – the system can receive mail, act as a FTP server whilst unattended.

TransSys PNI doesn't have a GUI-based installation process – it is necessary to actually edit configuration files to effect a working installation. In some cases, it is necessary to create new TCL script files if you are using a modem or a network access server for which a script does not already exist. Adding a NeXT-specific GUI front-end would significantly extend the development effort (which has dragged on *way* to long as it is), as well as increase the cost of the product. It also detracts from one of the prime motivations of this software - to be much more portable, both to multiple NEXTSTEP hardware platforms as well as to other UNIX-like operating systems.

The same scripting capability which is so powerful and flexible also makes PNI difficult to install with just a single mouse-click. PNI is more *expert-friendly* rather than *user-friendly*. There really is *a lot* that you can do; the challenge has been writing it down and documenting it..

So, is PNI for you? In many cases it may not be. It really is necessary in some cases to understand how IP networking actually works and how the packets fly. To really determine if PNI will solve *your* problem, get the free SLIP-only version see Section 1.2 [PNI package versions], page 2, and try it for yourself. Your only investment is your own time, which is certainly valuable enough.

## 1.2 PNI versions

The PNI software is available in 4 different versions: Each version consists of some set of enabled encapsulator types which defines the available set of functions which can be used by the **pnid** process.

- PNI-SLIP – A freely available version which implements SLIP. This version is available free of charge via anonymous FTP from FTP.UU.NET as /vendor/TransSys/TransSys-PNI-1.13.tar.gz, and from the usual NeXT Internet FTP archive sites (and eventually on CDROM collections). This version is available as a public service to the NeXT community - NEXTSTEP systems should have a SLIP solution available free of charge, as do many other platforms. (It also makes a great way to get in the door to demonstrate the product!)
- PNI-BASIC – a commercial version of the software which implements SLIP (as above) as well as CSLIP, which is SLIP with TCP Header Compression.
- PNI-PRO – a commercial version, which, along with the PNI-BASIC features, also includes IP packet filtering including idle link timeouts and support for IP *tunnels*.
- PNI-Enigma – a commercial version, which, along with the PNI-PRO features, also include the SECURE encapsulator which can be used to ensure that traffic has not been tampered

with during its transit over the network. This can be done by employing one of a number of algorithms to detect tampering (such as the MD5 Message Digest algorithm) or to prevent tampering or capture (such as a data encryption algorithm such as DES or IDEA).

The SECURE encapsulator is most commonly used with the IP TUNNEL encapsulator when sending sensitive traffic over "public", unsecured networks. This version is not available for export outside the United States.

The "assurance" algorithms are implemented in different Objective-C classes which conform to the see Section 12.3 [TamperProof], page 63 protocol.

*In fact, PNI-Enigma is not be available for the indefinite future. It may be that is will be necessary for TransSys, Inc. to register as a manufacturer of defense articles and obtain additional licenses to sell products which are defense articles. Sigh.*

All versions of TransSys PNI require NEXTSTEP release 3.1 or 3.2, and should function on both Motorola and Intel based systems.

Support of HP-PA and other platforms will be added when access to suitable development and test platforms can be arranged.

## 1.3 About this documentation

This documentation was prepared using the GNU *Texinfo* package, which allows one source document to be used for creating on-line help and reference material as well as a printed reference manual. It allows the author to use cross references and hypertext links to other parts of the document, which can be automatically followed if the document is viewed with a suitable user interface.

The Texinfo source is also processed by the *texi2html* PERL script to product HTML files suitable for use in a WWW server.

The documentation available to end users of the PNI software is available in three forms:

- PostScript, suitable for printing on most PostScript compatible printers. This document only uses these Type-1 PostScript fonts: *Times-Bold*, *Times-Roman*, *Courier-Bold* and *Courier*.

  You can also view the PostScript form of the file using the **`Preview`** application on NEXTSTEP platforms; it works best if you "Zoom In" once. On Motorola-based NEXTSTEP hardware with a 1120 by 832 display resolution, an entire page of text (other than headers and footers) is visible at once. The PostScript version of the documentation is provided in the PNI distribution.
- GNU **`info`** format, which can be browsed using emacs or by the standalone **`info`** program. The info files, '**`pni.info-*`**' are ASCII text can can also be viewed using any sort of text editor. The info format of the documentation is provided in the PNI distribution.

  The info format files contain the hypertext links, which can be followed using either the info reader in the emacs editor or the standalone info reader program.
- HTML (Hypertext Markup Language) format, as used in the WorldWideWeb (WWW) project. This version is not usually distributed with the PNI software, but is currently available from the TransSys WWW server, starting at the URL:

  **`http://www.TransSys.COM/TransSys/PNI/pni-info.html`**

This host is on a low-bandwidth SLIP link, so fetching copies of the documentation this way is not the best option. It is handy, however, for browsing the manual and checking for new versions, etc.

NEXTSTEP users can use the **OmniWeb.app** application to access WWW servers on the Internet. It is available via anonymous FTP from

**ftp.omnigroup.com: /pub/software/OmniWeb.app.tar.gz**

## 1.4 TCL documentation

Since much of the user-visible configuration and customization of the PNI package is done using TCL, some additional documentation is provided on TCL and some extensions to TCL, TclX. These files will exist in the '**doc**' sub-directory of the PNI distribution. (see Section 3.2.2 [File Archive Distribution], page 16).

Two files, '**doc/Tcl.man**' and '**doc/TclX.man**' are UNIX **man**-formatted files. You can examine these from a Terminal window:

```
nroff -man Tcl.man | more
```

or, to print a nicely formatted version on the default (PostScript) printer:

```
ptroff -man Tcl.man
```

## 1.5 TransSys DialUp-IP, the *other* TransSys SLIP

The TransSys PNI product began as a re-implementation of the existing and popular TransSys DialUp-IP SLIP/CSLIP software package. That software package enables the NeXT user to participate in a wide-area network via serial connections on the NeXT. This brings to the NeXT such applications as telecommuting as well as part-time, low-cost Internet connectivity. That package, which has enjoyed wide popularity in its free, demo version (SLIP) as well as a low-priced commercial product (CSLIP), needed some enhancements to respond to specific requirements from customers and potential new customers. The original design goals for the follow on product included support for the Point-to-Point Protocol (PPP) defined by the Internet Engineering Task Force (IETF) as well as support for hardware devices other than the NeXT's two serial ports (such as TTYDSP and various serial port expansion products).

The scope of the features which needed to be made available in a future product made a reimplementation rather than a simple retrofit of the existing software an attractive option. While adding new capabilities, a number of implementation issues in the first product could be addressed in the light of considerable operational experience. Among these are to reduce the impact of future NeXT operating system releases as well as having the software be portable to other vendor platforms.

It became clear that a reimplementation set the stage for entirely new and unique capabilities which have never been available before in a software package of this type. Existing SLIP and PPP serial networking software for workstations tend to be implemented in a monolithic fashion, with limited flexibility. The architecture for the TransSys PNI product is very modular and allows for expansion to include other functions not originally part of the package.

## 1.6 Acknowledgments

### 1.6.1 Credits

The TCP header compression feature is derived from software written by Van Jacobson. The MD5 message digest implementation uses the RSA Data Security, Inc. MD5 Message Digest Algorithm.

### 1.6.2 Copyright Information



PNI uses *TCL*, the Tool Command Language an extension language designed to be embedded into applications, as the basis for the control of the software. The complete source code to the base **TCL** package is available via anonymous FTP from **ftp.cs.berkeley.edu** in the '**/ucb/tcl**' directory.

```
/*
 * tcl.h --
 *
 * This header file describes the externally-visible facilities
 * of the Tcl interpreter.
 *
 * Copyright 1987-1991 Regents of the University of California
 * Permission to use, copy, modify, and distribute this
 * software and its documentation for any purpose and without
 * fee is hereby granted, provided that the above copyright
 * notice appear in all copies.  The University of California
 * makes no representations about the suitability of this
 * software for any purpose.  It is provided "as is" without
 * express or implied warranty.
 */
```

The scripting capability that's used to dial modems and log into remote systems is based on a heavily modified part of the *expect* program. The expect program also uses TCL as the basis of its scripting function.

```
/* expect.c - expect and trap commands
 *
 * Written by: Don Libes, NIST, 2/6/90
 *
 * Design and implementation of this program was paid for by
 * U.S.  tax dollars.  Therefore it is public domain.  However,
 * the author and NIST would appreciate credit if this program
 * or parts of it are used.
 */
```

Some useful extensions to TCL are also included; these are a subset of the Extended TCL package, *TclX*. Complete source code the the TclX package is available via anonymous FTP from **ftp.NeoSoft.COM** in the '**/pub/tcl/distrib**' directory.

```
/*
 * tclExtend.h
 *
 *    External declarations for the extended Tcl library.
 *--------------------------------------------------------
 * Copyright 1992 Karl Lehenbauer and Mark Diekhans.
 *
 * Permission to use, copy, modify, and distribute this
 * software and its documentation for any purpose and without
 * fee is hereby granted, provided that the above copyright
 * notice appear in all copies.  Karl Lehenbauer and Mark
 * Diekhans make no representations about the suitability of
 * this software for any purpose.  It is provided "as is"
 * without express or implied warranty.
 */
```

The packet filtering capability is based on the *Berkeley Packet Filter* and the **`tcpdump`** program, which were heavily modified to work as part of another application. The **`tcpdump`** program is available via anonymous FTP from **`ftp.ee.lbl.gov`** as '**`tcpdump-2.2.1.tar.Z`**'.

```
/*-
 * Copyright (c) 1990-1991 The Regents of the University
 * of California.
 * All rights reserved.
 *
 * This code is derived from the Stanford/CMU enet packet
 * filter, (net/enet.c) distributed as part of 4.3BSD, and
 * code contributed to Berkeley by Steven McCanne and Van
 * Jacobson both of Lawrence Berkeley Laboratory.
 *
 * Redistribution and use in source and binary forms, with
 * or without modification, are permitted provided that the
 * following conditions are met:
 *
 * 1. Redistributions of source code must retain the above
 * copyright notice, this list of conditions and the following
 * disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above
 * copyright notice, this list of conditions and the following
 * disclaimer in the documentation and/or other materials
 * provided with the distribution.
 *
 * 3. All advertising materials mentioning features or use
 * of this software must display the following acknowledgement:
 * This product includes software developed by the University
 * of California, Berkeley and its contributors.
 *
 * 4. Neither the name of the University nor the names of
 * its contributors may be used to endorse or promote products
 * derived from this software without specific prior written
 * permission.
 *
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS
 * ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING,
 * BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN
 * NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR
 * ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
 * AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN
 * IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 *
 * @(#)bpf.c      7.5 (Berkeley) 7/15/91
 */
```

This package contains an implementation of of the MD5 Message digest algorithm which is *"derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm"*. The MD5 digest algorithm is described in Internet *RFC-1321*, which also includes a reference implementation in the C programming language. This RFC can be obtained from a variety of different sources, including via anonymous FTP from **`RSA.COM`** as '**`/pub/md5.txt`**'.

## 1.7 Summary

The TransSys PNI product is a new approach to solving the SLIP or dial-up connectivity problem on the NeXT platform. The product was developed with portability in mind and will be available on Intel-based NEXTSTEP platforms as well as Motorola 68030 and 68040 platforms. Ports to other hardware platforms running NEXTSTEP, when they are announced, will also be performed very easily.

The modular nature of the product allows it to address new situations and applications easily, without major redesign of the software. For instance, conversion from SLIP to PPP is simply a matter of substituting the PPP loadable encapsulator for the existing SLIP/CSLIP encapsulator. Similarly, use of ISDN rather than a serial port requires only an ISDN encapsulator (and the requisite ISDN hardware, of course) and no changes to any of the other encapsulators such as the DES encryption or FILTER encapsulators. You can pick and choose the capabilities and features that your application requires. Want to see how your network based application functions over a wide-area, congested network? Write an encapsulator that randomly drops some percentage of the traffic passing through it. Vary the drop rate remotely and watch the effect on your application. You don't even need a network connection; use the LOOP encapsulator and simulate it all locally!

This sort of mix-and-match architecture will adapt to meet you needs. The TCL based configuration and scripting language is the most flexible you will find in any SLIP or PPP like product.

# 2 Release Notes

This manual is not yet completed and is in pretty bad shape at the moment. The information present is probably not organized in the best possible way. The emphasis thus far is to get what information is available written down, organization and presentation has suffered as a result.

## 2.1 Beta Test Software

Previous versions of PNI software were available for beta testing purposes. You should ensure that you are not using one of these version instead of the current release version.

## 2.2 Release History

### 2.2.1 1.4 Beta Release

This was the first beta release widely available.

### 2.2.2 1.5 Beta Release

This was an interim release, not available very long, which resolved a number of rather silly problems that crept into the distribution at the last minute. These were mainly confined to a number of TCL script files. Also a number of the modem dialing scripts were updated.

### 2.2.3 1.6 Beta Release

This version corrects a rather serious bug which, while benign and not noticed on Motorola 680x0 hardware, causes **`pnid`** on Intel 80486 systems to dump core due to a segmentation violation.

### 2.2.4 1.7 Beta Release

- Updated this document with more documentation describing the PNI software, including the **`dialer expect`** command. Also describe how to set up server mode and how to debug dialing scripts. Still need to thoroughly index the document.
- Added additional information which can be returned by the remote encapsulator query facility (via the **`pnitcl`** program). This isn't currently used except in debugging and development.
- When processing key registrations in the **`pnid`** program, resign any set-UIDness. Don't emit prompts for registration string if the standard input isn't a tty device.
- Add new TCL command, **`sysarch`**, which returns a two element list list containing the operating system and hardware architectures.
- In the PNI encapsulator, don't emit weird error messages when passed a "runt" packet for input to the kernel. Check for minimum sized packets and complain if they look broken.

- In the SLIP encapsulator, declare some data structures **`const`** to move them into the (read-only) code segment. When input packets exceed the configured MTU of the interface, look for the end of the bloated packet and emit the length in the "packet too large" error message:

  ```
  slip slip: rec'd frame len=%d exceeds mtu=%d
  ```

  If in auto-CSLIP mode, log a message when switching from uncompressed to compressed SLIP mode.
- In the TTY encapsulator, fix log message to specify the name of the network interface, rather than just **`pni`**.
- Added support/login-netblazer.tcl, login script for a Telebit Netblazer.
- In support/init.tcl, add some additional comments describing the use of the **`remcmd`** function to handle remote commands. Add the **`status`** command for remote queries to return a summary.

  Make some debugging messages condition on *testMode* being defined when **`pnid`** is invoked with the **`-t`** option.

  Add support for a summary file which logs start and stop entries *only* to determine use of PNI. File name is specified by the **`Config(callLog)`** configuration variable.
- Fix bogus comments in the support/login-annex.tcl file.
- Fix broken code in the support/login-unix.tcl file.
- Change the loadable kernel driver and installation scripts to use a different character special major device number. This has been assigned to TransSys, and should eliminate the incompatibility with Novell IPX.
- Fixes to the **`pnistat`** script, including a new command to return the process id of a particular instance of the **`pnid`** daemon process. It is also now possible to query more than one instance of the daemon in one invocation of the script.
- The install scripts should be a bit less prone to complaining about device conflicts when PNI is re-installed on the same host.
- Added (yet unused) *Feature* array to **`pnid`** which can easily be set by a command line option.

### 2.2.5 1.8 Beta Release

- The TCL interpreter used by PNI has been upgraded to the latest version, TCL 7.3. The corresponding version of the TCL extensions package, **`TclX`** has also been upgraded.
- A bug in the inactivity timer of the FILTER encapsulator has been fixed. The timer wasn't started until the first packet which matches the inactivity timer reset filter was seen, rather than when the encapsulator was initialized.
- *Important:* only config files of the form '**`/etc/pni/config/foo.config-auto`**' will be automatically started at boot time. This is a change from the previous version.

### 2.2.6 1.9 Beta Release

- Fixed a bug which required that a file '**/usr/local/tclX/7.3a/TclInit.tcl**' exist on the machine which the PNI package was installed on. This was a side-effect of having a pathname compiled into the TclX software. This is now overridden, and uses '**/etc/pni/TclInit.tcl**' instead. This bug caused the **pnibusy** program to not function.
- Reworked the '**/etc/pni/pnirun**' shell script to correctly handle the convention used to specify configuration files which are invoked automatically when the system boots. The behavior described for the previous release, see Section 2.2.5 [1.8 Beta Release], page 11, now actually works!
- Repaired a bug which kept the SLIP-only, non-expiring license key from working. *Important:* this change unfortunately will cause all of the license keys which have been previously issued to now be considered invalid, and unusable with the 1.9 Beta and later releases.
- Added support to the **makeinfo** program, the **texinfops.tex** TeX texinfo macro package, and **texi2html** for a new set of **@**ifhtml and **@**end ifhtml directives.
- The demo license key strings no longer include the **PPP** feature. This was only specified for future use; there *is not* currently PPP encapsulator available. Sorry for getting your hopes up.
- Fixed the priority queuing in the loadable kernel driver to convert TCP port numbers from network byte order (*"big-endian"*) to host byte order when checking for traffic to and from "interactive" services. This wasn't working on Intel-based systems which have the wrong (*"little-endian"*) byte order.
- The TclX package has been upgraded to TCLX 7.3a patchlevel 2. TclX is a package of useful extensions to the TCL interpreter, which is used extensively in the PNI package. More information on TclX is available on harbor.ecn.purdue.edu in '**pub/tcl/extensions**'.

### 2.2.7 1.10 Beta Release

- The **pnitcl** program wasn't printing prompts in interactive mode; this was due to changes in the most recent version of TclX. **pnitcl** now prints a prompt composed of the program name and the history list item number.
- The **pnid** program no longer performs a **setpriority(PRIO_PROCESS, 0, -20)**. The daemon runs at "normal" priority, which should reduce the impact on the rest of the system. You can cause it to revert to the previous behavior by adding the **-o priority=-20** to the **pnid** invocation.
- Add **-debug** and **-down** options to the **pnirun** shell script. The **-debug** option can be used to run a **pnid** daemon process interactively while debugging dialing and login TCL scripts.

  The **-down** option can be used to halt a named instance of the **pnid** process which is running.
- Documentation restructuring - moved supporting information into an appendix.
- Fixed a bug in the packet decoder/dumper subroutine which inadvertently applied ntohs() to the ICMP type and code fields, which are only 8 bits wide.
- New utility TCL function, **printify**, defined which will convert carriage return, line feed and tab characters in a strings to printable representations. This is useful since messages which are

logged using the syslog facility are truncated at the first newline character. The **log**, **dlog**, and **warn** TCL functions are updated to use **printify** to clean up the message text to be logged.

- The **pnistat** TCL script has been updated to be a little more graceful when invoked and no daemon is running. Rather than a many line TCL traceback being dumped, a simple one line warning message is emitted instead.

### 2.2.8 1.11 Release

- Converted TCL scripts to use the **switch** primitive rather than the depricated **case** primitive.
- Fixed error message in the **pnibusy** script.
- For hard-wired serial lines, added the ability to have **pnid** ignore the state the the DCD (Carrier Detect) RS232 control signal. The use of this feature is *not* recommended since you cannot detect when a modem connection has dropped, which is signified by the modem dropping the DCD signal. It is useful, however, when a hardwired connection exists between two directly connected computers and the cable doesn't correctly driver one computer's DCD signal from other other computer's DTR signal as a correct "null modem" cable should.

  Check the documentation on the TTY encapsulator for information on how to configure this features.
- The **pnid** daemon now correctly responds to the null RPC call. Now, **rpcinfo -u localhost pni0** works as expected.
- Initialize syslog a bit earlier to catch errors during early initialization activities.

### 2.2.9 1.12 Release

- The **pnid** daemon will now more gracefully deal with failures during the phase1 initialization, and not attempt to perform phase2 initialization procedures. This means that if the network interface is already active and in use by another daemon, further initialization will not be attempted.
- If an error occurs in the '**PNI.encap/PNI_objInit.tcl**' file containing initialzation scripts for the PNI encapsulator, then don't try to log the contents of the **errorInfo** variable if it's not set. Previously, this resulted in a another TCL error traceback which was confusing.
- And the previous error occured because the **errorInfo** variable wasn't declared **global** in the TCL procedure.
- We now periodically reset the the priority of the pnid process. This is because it tends to get "auto-niced" after it's been running for a while. This causes performance to suffer somewhat, and manifests itself as not as snappy remote-echo performance.
- Fix error in the '**support/login-pni.tcl**' script which incorrectly referenced an incorrect variable when trying to log an error message.
- The kernel driver will now mark the network interface as "down" when the associated **pnid** daemon process exits and closes the pseudo network interface kernel driver.
- The kernel driver no longer (incorrectly) identifies itself as Beta Test Software.

- Fix a bug in the Installer's post_install script which causes a literal "*" to inadvertantly get file globbed. This happens when creating new groups using the **niutil** program.

### 2.2.10 1.13 Release

- The **pnibusy** script is no longer used. It should now be possible for anyone in the **pni** group to start an instance of the **pnid** daemon without running as root.
- Fix various login scripts to not use the TCL **file readable ...** construct, it TCl uses the **access()** system call internally. This doesn't do the Right Thing since access() check file access based the *real* UID of the process, rather than the *effective* UID. Since **pnid** runs setuid to root, this causes things not to work correctly.
- The PNI encapsulator now keeps track of which routes it actually succeeded in installing (a default route and/or a route for the local address of the PNI interface via the loopback interface), so it can properly clean up after itself.

## 2.3 Future Plans

These are some of the feature which are not currently implemented:

- First, and most obvious: the documentation is inadequate for use by mere mortals.
- The PNI software uses a UNIX character special device to communicate with the kernel-level pseudo network interface driver See Section A.3 [**pni_reloc** kernel driver], page 81. This is currently hardcoded to attempt to use a major device number of 29. On Motorola-based NEXTSTEP systems, it is necessary to use a fixed major device number, while on Intel based systems, it is possible to have one dynamically assigned. The dynamic assignment is not yet done on those systems, so a potential for conflict exists. The major device number used by PNI was assigned by NeXT to TransSys exclusively, Inc. so there should be minimal opportunity for conflict.
- Control and monitoring is yet somewhat minimal. The PNI software uses Sun RPC to communicate to the daemon process. There is a program to initiate the RPCs (via a TCL primitive); the daemon process will receive them and authenticate their origin using an MD5 digest over the message and a password.

  There is a TCL script provided, '**/etc/pni/bin/pnistat**' which uses a TCL interpreter augmented with PNI specific commands (**/etc/pni/bin/pnitcl**), to extract statistics from instances of the **pnid** daemon.

  One known problem currently is that whilst the daemon is in the midst of processing the modem dialing and login scripts, it will not respond to remote queries from the monitoring tools.

# 3 Installation

In this chapter, we'll examine briefly the steps required to install the PNI software on your NEXTSTEP computer system. We'll cover only the basic structure and steps of the installation and configuration process; the details will follow in later sections.

## 3.1 System requirements and prerequisites

The PNI package will run on NEXTSTEP versions 3.1 and 3.2 on both Motorola and Intel based platforms. Installing is requires approximately 1 MB of disk space.

A few issues to consider:

- 68030-based NeXT Cube systems do not have serial ports which support hardware flow control. This means that SLIP applications over a serial port will have to be run at no faster than the modem link speed, before any compression, for best effects.
- The distributed serial driver for Intel NEXTSTEP 3.1 systems is defective, and will cause the system to crash under moderate load. Upgrade to 3.2 as soon as possible.
- Motorola-based NEXTSTEP 3.2 systems using TTYDSP serial hardware will likely require a TTYDSP driver update, as the earlier versions would consume a great deal of CPU time when used by PNI.
- There is currently no support for HP NEXTSTEP platforms due to lack of available cross-development tools hosted on 68040 based NEXTSTEP systems, targeted to generate HP object code.

## 3.2 Installing the distribution

The software distribution containing the PNI software will consist of a NEXTSTEP Installer package. The package may have been provided to you on a floppy disk. Alternatively, you may have the file in its electronic distribution format in the form of a '**`.tar`**', '**`.tar.Z`**', or '**`.compressed`**' file.

To install and use the PNI package, the following tasks need to be completed:

1. Load the software distribution on the NEXTSTEP system from the distribution medium (or electronic distribution) using the Installer application.
2. Acquire the information needed to configure a network connection using the PNI package. This information include IP network addresses, the subnet mask, and routing information. In the case of dial-out SLIP or CSLIP connections, it is also necessary to determine what type of modem is to be used, the telephone number to be dialed, the type of remote network access server and any username and password information that's required to bring up the link.
3. Prepare a PNI configuration for each network connection. This is a file where the information that we collected is specified for the PNI software to use.

Subsequent sections in this chapter will detail the installation steps, and will also include a "quick start" to a common PNI configuration: a dial-up SLIP connection.

### 3.2.1 Before Installation

Before installing the PNI software, you need to consider a few installation choices.

First, the PNI software makes use of a UNIX group called **pni** which will be the group-owner for a number of files. By default, if no **pni** group exists at installation time, one will be created with a UNIX group id of 42. If this group id is already in use, the installation process will begin examining group id sequentially increasing group id values until it finds an available group.

If you would prefer to specify a group id, create a **pni** group using the **UserManager** application prior to installing the PNI software.

The other choice is to consider where you would like the software installed. By default, it will be installed in the '**/LocalApps/PNI**' directory. You may chose other locations for the software to be installed if you like; the installation script will create a symbolic link from '**/etc/pni**' to the actual installed location. The '**/etc/pni**' path name is the *only* path hard-coded in the PNI package; it is used to identify the location of the package. (Actually, the other hard-coded path names are the character special devices '**/dev/pni*.**')

### 3.2.2 Electronic file archive installation

If you have obtained an electronic file archive distribution, either by FTP from an archive site on the Internet, or perhaps via NeXTMail, or by some other download process then it is necessary to unpack the distribution archive so it can be installed. This process is necessary because the *archive* is used to group together (and compress to reduce the size of) a collection of files as a single file for transmission. The distribution will be an archive of a directory (or folder) called '**PNI-version**', where '**version**' will indicate which revision of the PNI software is being distributed. Within this folder will be a package file as well as some other support files such as a '**README**' or other auxiliary information.

From the a UNIX shell:

```
cd /tmp
gzip -d -c < /some/path/to/PNI-1.13.tar.gz | tar -xvf -
```

### 3.2.3 Diskette Distribution

You may have received the PNI software on a distribution disk; either on a 1.44MB floppy disk or perhaps on a CDROM. In this case, there is no file archive which need to be unpacked or unarchived; all of the requisite files are directly accessible. On a floppy disk, all of the files will be stored at the root of the file system. On a CDROM, the files will likely be stored in a folder corresponding to the PNI product.

### 3.2.4 Using the Installer Package

To install the package, you must run the NEXTSTEP '**Installer.app**' application. To do this, log into your system as the system administrator **root**, and double-click on the '**PNI.pkg**' icon in the Workspace. This will launch the the '**Installer.app**' application.

Now clock on the '**`Install`**' button. You will be given an opportunity to choose where the software will be installed. You can just use the default location ('**`/LocalApps`**'), or perhaps you can choose some other location (such as '**`/usr/local`**' or '**`/usr/pni`**'). **Caution:** *You should not install it as '**`/etc/pni`**'!* Later on, during the installation process, a symbolic link at '**`/etc/pni`**' will be created to point to the actual installed location; it is important to not preempt this file name.

You also have the option to choose which hardware architectures are to be supported. The PNI package is distributed as a "fat" binary that supports both Motorola 680x0 and Intel platforms. By default, only the current hardware architecture will be select (Intel for i486 and Pentium based systems, and Motorola for NeXT "Black Hardware". If you like, you can install *both* architectures, though you probably don't want to, as the package will consume more disk space.

Once you have decided on the location, click on the '**`OK`**' button. You will be warned by an Alert panel that installation programs will be run; click OK to continue.

The pre-installation installer program will indeed verify that you are running as **root**; please do not attempt to disable this check. Part of the installation process updates some system file and your local NetInfo domain. It is required that the user be **root** to perform these operations.

After the files have been installed from the Installer package to the local disk, the post-installation program will be invoked. The post-installation program will then continue and perform other tasks to complete the installation.

### 3.2.5 Installation-time system changes

A number of changes are made to your system by the post-installation program that's invoked automatically by the installer. You should be aware of the changes that are made so that you do not inadvertently un-do them in the course of normal system administration functions.

#### 3.2.5.1 The PNI package symbolic link

It is necessary to be able to reference the whole subtree of installed files after they are installed to locate various configuration files, and other files required for operation of the **`pnid`** daemon. Since the package can be installed anywhere in the UNIX file system, it is necessary to have an "anchor" somewhere that can be used to discover the installed location of the package.

When the package is installed, the post-installer program will create a symbolic link at '**`/etc/pni`**' which points to the installed package. All of the references to the installed files are done relative to this symbolic link, so it is important that this symbolic link be present and point to the correct location.

If the post-installer program discovers a pre-existing symbolic link, it assumes that it is an artifact of a previously installed version of the PNI package and replaces it. If '**`/etc/pni`**' is a plain file, then an Alert panel is used to query the user and determine if the file will be removed or if the installation will be aborted.

### 3.2.5.2 Device names in '/dev'

A number of character special device files will be created in the '**/dev**' directory. These are used to communicate with the loadable kernel device driver and are each one "half" of a pseudo network interface. There will be 16 files created in the '**/dev**' directory: '**/dev/pni0**', '**/dev/pni1**', . . ., '**/dev/pni9**', '**/dev/pnia**', '**/dev/pnib**', . . ., '**/dev/pnif**'.

The critical issue relating to the character special device files is which character special major device number is used. The major device number can be considered to be an index into a table that is used to reference different UNIX kernel drivers. Only a single driver can use a particular entry in this table, so it is essential that conflicts with existing drivers be avoided.

The PNI package uses character special major device number 29. The post-install program will examine the existing files in the '**/dev**' directory for conflicts; if any files are found that conflict, they are displayed and you are asked if you want to continue. If the files that are listed are '**/dev/pni0**', etc., then its likely that you are installing a new version of the PNI package, and there really is no conflict.

### 3.2.5.3 Changes to the rpcs NetInfo directory

The post-installation program will install a set of new *RPC* (Remote Procedure Call) program names. The PNI package uses Sun RPC to monitor and control the **pnid** programs that are running. Each **pnid** daemon registers for a different RPC program number which is used to select which daemon you are interested in monitoring.

A mapping exists between symbolic *program names*, which are mnemonic and hopefully easy to understand and remember, and the actual RPC *program numbers* which are used to rendezvous with the program willing to perform the RPCs. This mapping is stored in the local NetInfo database in the '**/rpcs**' location in the NetInfo database.

The program names that are used are '**pni0**' . . . '**pni9**', '**pnia**' . . . '**pnif**'. These correspond directly to the names of the network interfaces as well as the names of the character special device files created in '**/dev**'. (Although there is no requirement that they have this correspondence, it sure is handy!)

The program numbers that are used by the PNI package are in a range which start from 395250. This number range has been assigned by Sun Microsystems (who administer this space) for the exclusive use of TransSys, Inc. and our software. You should not experience any conflicts; if you do, we'd be very interested in hearing about it!

The post-installation script will use the **niutil** program to install the RPC mappings. You can examine these at any later time by using this command at the a UNIX shell:

```
nidump rpc .
```

you should see the following (though they may be in a different order);

```
pni0    395250
pni1    395251
pni2    395252
```

```
pni3    395253
pni4    395254
pni5    395255
pni6    395256
pni7    395257
pni8    395258
pni9    395259
pnia    395260
pnib    395261
pnic    395262
pnid    395263
pnie    395264
pnif    395265
```

#### 3.2.5.4 Changes to '/etc/rc.local'

The '`/etc/rc.local`' file is modified during installation so that the various **pnid** daemons can be started when the system is booted. The modifications that are made are of a very specific format that can be recognized at a later time and removed should you decide to uninstall the software package.

The following lines will be added to the '`/etc/rc.local`' file:

```
# PNI::START::
##
##  Danger! Warning!  Do not remove the
##          # PNI::START:: or the # PNI::END::
##  lines in this file.  The are used to automate
##  installation and de-installation of this
##  software package.
##
if [ -f /etc/pni/pnirun ]; then
  (/etc/pni/pnirun -all -boot -core & echo -n ' pnid') \
     >/dev/console 2>&1
fi
##
# PNI::END::
```

It is very important the the two delimiting lines

```
# PNI::START::
# PNI::END::
```

not be modified or removed. If they are, then it will not be possible for the PNI package to be completely removed automatically. Any other lines added between the delimiting lines will also be removed should you choose to de-install the package.

If you are going to use a TTYDSP serial port on your Motorola-based NEXTSTEP system, then it may be necessary to move these modification to the '**/etc/rc.local**' file to a place *after* the TTYDSP product is initialized.

#### 3.2.5.5 Changes to '/etc/syslog.conf'

The '**/etc/syslog.conf**' file is also modified so that log messages produced by the PNI package are collected together into on e log file for easy perusal. The following lines are automatically added to the '**/etc/syslog.conf**' file:

```
# PNI::START::1.0
local5.debug                    /usr/adm/pni.log
# PNI::END::1.0
```

You may wish to change the **debug** level specified to **info** or perhaps **notice** to see fewer messages once you have installed the package and it is functioning correctly.

In a similar manner to the '**/etc/rc.local**' file (see Section 3.2.5.4 [Changes to /etc/rc.local], page 19) the file changes are marked with distinctive delimiters to allow the change to be removed if the package is de-installed. Do not modify or remove the delimiting lines or this automatic removal will not function.

### 3.2.6 License Key Registration

The next step in the installation process is to register the software by using the *License Key String* supplied with the software. This information is used to register the software on the system it has been installed on.

When the registration panel appears, please fill in the required information including the License Key String. *Note:* to avoid confusion between characters which have similar appearance, the set of characters that are used to make the License Key string are restricted to:

- The 10 digits **0 1 2 3 4 5 6 7 8 9**
- The 25 UPPER CASE characters **A B C D E F G H I J K L M N P Q R S T U V W X Y Z**. Note that the letter **O** (oh) is not included so it will not be confused with the number zero.
- The 25 lower case characters **a b c d e f g h i j k m n o p q r s t u v w x y z**. Note that the letter **l** (ell) is not included in this set so that it will not be confused with the number one.
- The 4 punctuation characters **+ / @ %**

The License Key String will be at least 16 characters from the set defined above (and longer if it is a demonstration key with an expiration date). The License Key String, along with the user information supplied on the registration panel is used to construct a '**/etc/pni/keyfile**' file which the **pnid** daemon uses. Do not delete this file, or you will need to reregister the software.

Some distributions of the PNI software will have a '**/etc/pni/keyfile**' pre-installed. If this is the case, you will be asked during the installation process if you would like to keep the existing keyfile. You should answer *yes* to keep the existing demo keyfile.

## 3.3 Configuration Preview

Now that the software has been installed and registered, it is necessary to configure the software for your specific environment.

To perform this configuration, it is necessary to create a configuration file for each Pseudo Network Interface that you wish to use. In many cases, a single network interface for a SLIP/CSLIP connection is all that's required.

It may also be necessary to modify or create from scratch additional support files which are used to dial modems and "log in" to remote systems.

For network links which are originated or established from the local host, the configuration files are stored in '**`/etc/pni/config`**'. Support files are located in '**`/etc/pni/support`**'. (Recall that '**`/etc/pni`**' is a symbolic link to the installed directory.) Please note that files which you create or modify in the '**`/etc/pni/config`**' directory will not be modified or overwritten if you install newer versions of the PNI package.

For network links that are originated remotely, See Section 4.4.1 [PNI Network Access Server configuration], page 29, for more information.

### 3.3.1 Quick Start

There are three sections of information in the sample configuration files; your configurations will likely reflect this structure as well. Simply put there is:

1. The information needed by each of the components (encapsulators) of the **`pnid`** program. This section consists of a series of TCL **`set`** commands.
2. A list of the components (encapsulators) being used in this particular information. This section consists of a ordered list of TCL **`stack`** commands.
3. Procedure definitions for special purpose, custom actions to be performed when a link become active or goes inactive. This section consists of a series of TCL **`proc`** commands to define procedures to be invoked.

We'll briefly cover the sample configuration; detail information is available in later chapters of the manual, organized according to each encapsulator.

### 3.3.2 Tour through sample SLIP configuration

If you are just bringing up a remote SLIP or CSLIP link from your host, dialing some network access server, you can probably take an existing sample configuration and modify it for your use.

A sample configuration, found in '**`/etc/pni/SAMPLES/config.slip`**' should be copied to a file named '**`/etc/pni/config/pni0.config`**'. Note that if you want a **`pnid`** daemon started automatically at boot time, you must append **`-auto`** to the file file name, *e.g.*, '**`/etc/pni/config/pni0.config-auto`**'. See Section 13.4 [The /etc/pni/pnirun shell script], page 70, for details and more information.

For a bit more through commentary, look at the '**`/etc/pni/SAMPLES/config`**' file which has a bit more detail, or skip to the next chapter.

```
#
#  Pretty close to minimal configuration file for just
#  plain SLIP using a ZyXEL modem dialing into a
#  Xylogics Annex SLIP server.
#

## PART I.
```

First, we will specify the configuration information for the **pni** encapsulator which corresponds to the network interface and associated IP address information. We specify the name of the network interface which will be used ('**pni0**' in this case); a network interface can only be used by one **pnid** daemon process at a time. The MTU (Maximum Transmission Unit) of the interface is also set; for plain SLIP connections, this usually defaults to 1006 bytes, though some remote systems used as dial-in network access servers use 1500 bytes.

```
set Config(pni:INTERFACE)       pni0
set Config(pni:MTU)             1006
```

Now it is necessary to specify the IP address information. A SLIP connection is implemented using a point-to-point network interface; this type of interface is defined to have a *local* address (for your host) and a *remote* address (for the other end of the point-to-point link). A subnet mask should also be specified to indicate what part of the address is the *network* part, and what part is the *host* part. You network administrator should be able to supply you with this information.

```
set Config(pni:ADDRESS)         229.230.11.211
set Config(pni:REMOTEADDRESS)   229.230.11.111
set Config(pni:NETWORKMASK)     255.255.255.0
```

If you'd like a default route installed which uses this interface when the link is established (and removed when the link is dropped), then include the next line. If you don't want a default route dynamically installed when the link is established, omit this line

```
set Config(pni:DEFAULT)         1
```

The configuration for the '**slip**' encapsulator is rather simple: you only specify the MODE of the encapsulator which can be either '**slip**' or '**cslip**' (if your License Key authorizes use of CSLIP).

```
set Config(slip:MODE)           slip
```

Now we specify the configuration for the '**tty**' encapsulator; its job is to talk to a serial tty device to send and receive characters. We specify what device to use, and what speed the port should be set to use.

```
set Config(tty:DEVICE)          /dev/cufa
set Config(tty:SPEED)           38400
```

We also indicate what type of modem is connected to the serial port and what type of network access server will be called. The names that are used refer to support scripts in the script search path, which are the following directories: '**/etc/pni/config**', '**/etc/pni/support**', '**/etc/pni/TTY.encap**'. So, if you need to modify one of the scripts which are provided by the PNI software, place them in the '**/etc/pni/config**' directory where they will *not* be deleted or over-written should you install a new version of software.

The '**MODEMTYPE**' specification refers to the
'**/etc/pni/support/dial-*.tcl**'
files, while the '**SERVERTYPE**' specification refers to the
'**/etc/pni/support/login-*.tcl**'
files.

If a dialer script or login script doesn't exist, you can create your own modeled on the existing scripts, name them '**login-foo.tcl**' or '**dial-bar.tcl**' and install them in the '**/etc/pni/config**' directory.

```
set Config(tty:MODEMTYPE)        zyxel
set Config(tty:SERVERTYPE)       annex
```

Now it is necessary to specify some information required by modem dialing and network access server login scripts. First, we specify the name of the network access server that we'll be dialing into. There is nothing special about the name that is picked, it is used in a subsequent configuration specification so it must be consistent .

Then the phone number to be dialed, and how the dialing is to occur (either **TONE** or **PULSE**).

```
set Config(tty:SERVERNAME)       yourSlipServerName
set Config(tty:NUMBER)           555-1212
set Config(yourSlipServerName:DIALTYPE) TONE
```

Finally, if the network access server you are connecting to requires you to "log in", then you must supply the user name and password for the '**/etc/pni/login-foo.tcl**' script to use. There are two ways to do this. A file name can be specified which contains TCL commands to set the username and password:

```
set Config(yourSlipServerName:SECRETFILE) \
                "/etc/pni/config/password.pni0"
```

The file '**/etc/pni/config/password.pni0**' would contain two lines:

```
set username sliplogin37
set password theSecretWord
```

The motivation for this option is that the regular configuration would contain no sensitive information and could be readable by anyone. The sensitive username and password could be kept protected in its own file, and only readable by the "root" user. In fact if you use a separate file, you *must* ensure that the file it not readable by "group" or "other" based on the UNIX permissions set on the file. This is checked for by the **pnid** scripts and an error will occur to notify the administrator that he may have inadvertently allow sensitive information to be exposed.

Alternatively, it is possible to encode the the user name and password in the configuration file. This is a bit simpler to configure, and may be appropriate in some situation where having the main configuration readable is not necessary.

```
set Config(yourSlipServerName:USERNAME)  sliplogin37
set Config(yourSlipServerName:PASSWORD)  theSecretWord
```

The next command set a password which is used to control what users have monitoring and control access to the **pnid** daemon process.

```
set Config(CMD:password)         "The secret word"
```

If this is omitted, then a password that was randomly generated at installation time will be used instead. The random password is in the '**/etc/pni/config/password**' file, and readable only by root and the **pni** UNIX group. This is the preferred mechanism.

The next part of the configuration indicates what encapsulators will be used together to connect this particular network interface to the network. The specification of encapsulators are done in order, starting from the host's logical network interface and working towards the physical network connection.

The specification of encapsulators is done using the **stack** command. This command takes two arguments:

- The *type* of encapsulator. This is one of a few predefined names, and are always in UPPER CASE. These names correspond the the '**/etc/pni/*.encap**' directories.
- The *name* of the encapsulator. This name is use to refer to a particular instance of an encapsulator. Although not used in this particular configuration, it is possible to use an encapsulator type more than once; each would have a unique name. The encapsulator name is reference in the **set** commands in the first section of the configuration.

The very first encapsulator is *always* the '**PNI**' encapsulator. The very last encapsulator is what actually talks to the "hardware"; the physical connection to the network. In between are the encapsulators that take you from one end to the other.

For a SLIP application, the sequence of '**PNI**', '**SLIP**', '**TTY**' is what is required: '**PNI**' will hand complete IP packets to the '**SLIP**' encapsulator. '**SLIP**' will take IP packets, and perform the SLIP-protocol "byte-stuffing" and framing, resulting in a stream of characters. The '**TTY**' encapsulators takes streams of characters and puts them on the serial port.

The same sort of operation takes for input packets. The '**TTY**' encapsulators hands a few characters at a time (as they arrive on the serial port) to the '**SLIP**' encapsulator. '**SLIP**' accumulates characters until it receives an entire frame, decodes this to an IP packet, and then hands it to '**PNI**'. The '**PNI**' encapsulator then causes the IP packet to be handled as input to the kernel's network interface.

```
# PART II.

stack PNI       pni
stack SLIP      slip
stack TTY       tty
```

There is a "global", non-encapsulator specific configuration variable called **callLog**, which specifies the name of a file which will receive logging records each time the link is established and broken.

```
set Config(callLog) "/usr/adm/pni.call-log"
```

The records logged look like this:

```
STOP  pni0 Mon Mar 14 22:01:26 1994 - Mon Mar 14 22:04:18 1994
START pni0 Mon Mar 14 22:05:12 1994
STOP  pni0 Mon Mar 14 22:05:12 1994 - Mon Mar 14 23:01:54 1994
START pni0 Mon Mar 14 23:07:23 1994
STOP  pni0 Mon Mar 14 23:07:23 1994 - Mon Mar 14 23:17:47 1994
START pni0 Tue Mar 15 10:32:07 1994
```

```
STOP  pni0 Tue Mar 15 10:32:07 1994 - Tue Mar 15 12:37:19 1994
START pni0 Tue Mar 15 12:42:34 1994
STOP  pni0 Tue Mar 15 12:42:34 1994 - Thu Mar 17 16:42:35 1994
```

Finally, there are two TCL “hooks” that can be defined which can perform specific actions as events occur. The two events currently available are a “start” event and a “stop” event. The correspond to when the last encapsulator in the list establishes a link (usually the TTY encapsulator). You can cause any local actions to occur here that you with. One possibility is to run the sendmail queue when the link comes up by using the **`exec`** TCL command to run a program (if you do this, make sure the program run is put in the background).

```
# PART III.

proc LINK_start { encap } {
    log "LINK $encap connected"
}

proc LINK_stop { encap } {
    log "LINK $encap disconnected"
}
```

More detailed information on how each encapsulator can be configured follow in the next chapter.

# 4 Configuration

Configuration of the PNI package is done using *TCL*, the Tool Control Language. TCL is a procedural embedded extension language; it supplies all of the control structures, data structures, and a set of standard commands. TCL is then extended by adding application specific command which can be invoked.

It is instances of the **pnid** daemon which need to be configured; the loadable kernel driver that implements the Pseudo Network Interfaces is generally not customized or configured. It is possible, under certain circumstances to change some operational parameters of the loadable kernel driver; particularly which character special major device number it uses (see Section A.3 [pni_reloc kernel driver], page 81).

All of the interesting functions of the **pnid** program lie in the individual encapsulators. The basic **pnid** program serves as the supporting framework to allow these encapsulators to function and be controlled. It should be of no surprise that almost all of the configuration information required pertains to the individual encapsulators, rather to the **pnid** program itself.

The configuration information for a particular instance of a **pnid** program is stored in a plain ASCII text file. In the case where the local host is actively initiating the link, the configuration files are stored as '**/etc/pni/config/pni***unit***.config**', where *unit* is used to distinguish between different interfaces ('**pni0, pni1,**' etc). The **pnirun** shell script will automatically use these files at boot time (see Section 13.4 [The **pnirun** script], page 70).

The configuration file consists of lines of text which are taken to be a series of TCL commands to be invoked as the **pnid** program is initializing itself.

In addition to the configuration file, two other files of TCL commands are also loaded and run by the **pnid** program. These files are '**/etc/pni/pnid.tcl**' and '**/etc/pni/support/init.tcl**' which provide support. These files consist mainly of TCL procedure definitions.

Other TCL files that are loaded by the **pnid** program are specific to the encapsulators which are being used. Encapsulators are Objective-C loadable classes, and reside in *bundles* stored in the '**/etc/pni**' directory. Each bundle consists of code that is dynamically loaded at run time as well as other supporting files; in this case, these files include two files that contain TCL commands. For example, consider the '**PNI**' encapsulator (which will be present in every configuration). The bundle directory is '**/etc/pni/PNI.encap**'; the actual machine code that implements the Objective-C class would be found in the file '**/etc/pni/PNI.encap/PNI**'. Supporting TCL commands would be found in the files '**/etc/pni/PNI.encap/PNI_Init.tcl**' and '**/etc/pni/PNI.encap/PNI_objInit.tcl**'. The difference between the two files is that the first file is loaded only once if the encapsulator is in use. The second file of TCL commands is loaded once for each instance of the encapsulator; it may be that the same encapsulator class is used more than once, with each configured differently.

## 4.1 Encapsulators

The operation of each of the encapsulators is configured by invoking encapsulator specific TCL commands. Normally, these commands are not from the "simplified" (!?!) configuration file that's normally used. The normal course of events is that only elements in the *Config* TCL array are specified by the user; the encapsulator specific TCL commands are invoked by configuration procedures that operate based on the user specified configuration information.

At initialization time, a TCL procedure is invoked for each encapsulator. For example, if the SLIP encapsulator was used in a particular configuration, and it was "named" *slip*, the TCL procedure that would be invoked to configure that encapsulator would be called **slip_configure**. These **configure** procedures are normally defined when the encapsulator's various '**/etc/pni/*.encap/*Init.tcl**' files are loaded. These procedure, when invoked, examine the defined values in the *Config* array, and perform the required low-level encapsulator-specific TCL commands.

In the following sections, the commands specific to each encapsulator are listed. In most cases, you can do what is required by using the simplified "Configuration Directives" that each encapsulator's **configure** procedure examines

## 4.2 Configuration Directive Conventions

The Configuration Directives are normally just specific values set in the the global TCL *Config* array. This array, like all TCL arrays, is subscripted by a string. The string subscript by convention consists of a prefix name, which refers to the encapsulator name defined on the stack command. The prefix name is followed by a colon and then the name of some configuration parameter.

```
set Config(slip:MODE)   cslip
```

This configuration naming structure is used since it may be possible to have more than one instance of an encapsulator used in one configuration (two SLIP encapsulators, for instance). Using this scheme, it is possible to have different instances of an encapsulator have much different configurations.

## 4.3 Configuration Primitive Conventions

The encapsulator's configuration primitives map directly to a C function or Objective-C method invocation. These primitives also have a particular naming structure. When an encapsulator (of a particular type) is instantiated (with a specified name). The name of this instance is used to create a TCL command. So, when making a configuration like:

```
stack   PNI     pni
stack   FILTER  filter
stack   SLIP    slip
stack   TTY     serial
```

Four additional global TCL commands are created, named: **pni**, **filt**, **slip** and **serial**. These commands are used in such a way that they expect a "sub-command" name to follow the command name. Each encapsulator inherits a set of "sub-commands" that are implemented by the encapsulator's abstract super-class. These sub-commands include **hashstats**, **commands**, **debug** and a few others.

For example, each instance of the PNI encapsulator has a TCL sub-command created called **mtu** to set the MTU of the interface. Assuming the PNI encapsulator was named '**pni**', the subcommand could be used like this:

```
pni mtu 1006
```

to set the MTU to 1006 bytes.

The configuration primitives associated with each encapsulator are not generally documented, as the user of the PNI software should not find it necessary to use them directly. Instead, a "higher-level" set of configuration variables is used, and TCL code specific to each encapsulator is responsible for invoking the configuration primitives as required.

## 4.4 Configuration as a Network Access Server

The PNI package can be used as a Network Access Server, to accept incoming sessions, rather than initiating sessions to a remote server. A network access server is a device which accepts some manner of connection from some remote user, and provides IP-level access to the network. That is, it acts as a router to switch IP packets between the just-established connection and the network it is connected to. This differs from a simple terminal server which expects *terminals* to be connecting with streams of ASCII characters being sent and received as part of a login session. Rather, it expects to transmit and receive entire packets from an IP capable *host*.

There are two major tasks that need to be performed to configure an incoming session. The first is to create a special user account for the incoming session. Then you must configure the **pnid** process, as usual, for that network interface.

The the special user account is needed so that when the remote system logs into the local system, an instance of the **pnid** daemon will be available to terminate the connection. This also allows the use of the serial port(s) for other purposes when they are not being used for a serial network connection. For example, FAX transmission and reception (PNI works well with the NXFax product), the UNIX **tip** program, and outbound PNI connections.

The user name for the account needs to be the same as the network interface to be configured for that link. So the user names would be named **pni0, pni1**, etc. The user name is used to pick the appropriate **pnid** configuration file to be loaded when the **pnid** program is started up for this connection.

The UNIX UID should be one assigned for use exclusively for incoming PNI sessions. All of the pseudo-users created can share the same UID value if desired. The UNIX group for the pseudo-user should be set to that group id for the *pni* group. This group was automatically added when the PNI software was installed. By default, the GID of 42 is used. Using the *UserManager* application, you can simply choose the *pni* group from the list as you create the user.

The home directory for the user can be set to '**/tmp**', since it is not really used. Another possibility, if you choose, may be '**/etc/pni/runtime**'.

The shell for the user must be '**/etc/pni/pnilogin**' which is a shell script that eventually invokes the **pnid** daemon with the appropriate configuration file.

The password for the account should be something that need only be known to the remote SLIP user, for use whilst completing the login process. Choose a "good" password which is not easily guessed.

The resultant password file entries that are created will look something like these:

```
pni1:jJjk3450sfj12:20:42:test SLIP login:/tmp:/etc/pni/pnilogin
pni2:4j912Jkl3894A:20:42:test SLIP login:/tmp:/etc/pni/pnilogin
```

(These can be displayed by using **nidump passwd .** from a Terminal window.)

### 4.4.1 PNI Network Access Server configuration

When the PNI software is used as a Network Access Server, the **pnid** daemon is invoked when the remote session is established, rather than running as a long-lived process. An account is created on the host acting as a Network Access Server corresponding; the account name should be named the same as the network interface to be used to communicate with the remote host.

For example, if a network interface *pni1* is to be used to communicate with a particular remote host, a local user also called *pni1* is created, with its login shell set to be '**/etc/pni/pnilogin**'.

A configuration file need to be created for the **pnid** daemon used in a server role, just as when it is used in its usual client role. A special naming convention needs to be followed when creating the configuration for use in server mode. In the case of an inbound connection for the pni1 interface, the name of the file needs to be '**/etc/pni/config/pni1.config-inbound**'.

he same sort of information need to be specified in the configuration file in server mode; one large difference has to do with the scripts that are used for *dialing* and *logging in* when in server mode.

Since the connection is established remotely, there is no dialing script required to establish the connection. When in server mode, the **PNI** encapsulator will the the *standard input* and *standard-output* of the process to receive and send characters on the serial connection.

There is a special login script: '**/etc/pni/support/login-pniserver.tcl**' which is used in conjunction with with the '**/etc/pni/support/login-pni.tcl**' script on the *other* end of the link (assuming that the other end is also using PNI). The purpose of the special login file is to emit certain bits of information that the remote system may be expecting, such as the fact that the PNI software is now running and ready; what the configured IP address of the local and remote systems are; and when SLIP operation actually begins.

## 4.5 PNI specific TCL commands

Beyond the standard set of commands and features available in the **TCL** scripting language, there are a number of commands available which are specific to the **PNI** package. These can be useful to authors of dialing scripts or anyone that needs to modify any of the TCL scripts provided which control the operation of the **PNI** package.

Most of the **PNI** specific **TCL** commands are specific to a particular encapsulator; those are described in section that documents the function and configuration of that particular encapsulator.

There are, however, a number of commands which are generally available which will be described here.

### 4.5.1 Logging `TCL` commands

**log** *Message-args...* TCL Command
: This command will log a message using the current **`syslog`** facility, at the logging level of LOG_INFO.

**dlog** *Message-args...* TCL Command
: This command will log a message using the current **`syslog`** facility, at the logging level of LOG_DEBUG.

**notice** *Message-args...* TCL Command
: This command will log a message using the current **`syslog`** facility, at the logging level of LOG_NOTICE.

**warn** *Message-args...* TCL Command
: This command will log a message using the current **`syslog`** facility, at the logging level of LOG_WARNING.

**syslog** *LEVEL MESSAGE* TCL Command
: This command will log a message with the severity level specified by **`LEVEL`**, which can be one of **`DEBUG`**, **`NOTICE`**, **`WARNING`**, **`ERR`**, **`CRITICAL`**, **`ALERT`** or **`EMERGENCY`**. The level can optional be prefixed with **`LOG_`**, so that **`LOG_ERROR`** is equivalent to **`ERROR`**.

  The **`MESSAGE`** argument is the string which is logged. Due to the implementation of the **`syslogd`** logging daemon, only the text up to the first newline character will be logged - the rest of the message text will be truncated. This is a consideration when logging text which has been matched in a modem dialer or login chat script, which may include multiply lines. There is a utility TCL procedure, **`printify`**, which will remove the embedded newline, carriage return and tab characters and replace them with a printable representation.

### 4.5.2 `pnitcl` specific TCL commands

The **`pnitcl`** program is a support program which included a TCL interpreter with the standard set of TCL commands, and the TCLX extensions. It also has a set of commands to perform RPC operations to running instances of the **`pnid`** program.

**cmdopen** *hostname rpc-program-number* TCL Command
: Creates an RPC client handle to perform commands to the **`pnid`** daemon registered as RPC program '**`rpc-program-number`**' on host '**`hostname`**'. Returns a handle that is passed to the **`cmdrpc`** and **`cmdclose`** commands.

**cmdrpc** *handle command* [*authinfo*] TCL Command

**cmdclose** *handle* TCL Command

Frees the resources associated with the `handle` specified and makes the handle invalid for any future uses.

# 5 **PNI** Encapsulator

The **PNI** encapsulator manages the operation of the virtual network interface. As such, it communicates with and configured the PNI kernel resident driver.

## 5.1 Configuration Directives

**Config(pni:INTERFACE)** Configuration Variable

The **INTERFACE** parameter is used to select which kernel pseudo network interface is to be used by this instance of the **pnid** daemon process. The parameter will usually be '**pni0**' to use the first pseudo network interface.

**Config(pni:MTU)** Configuration Variable

Each network interface has an associated *Maximum Transmission Unit* associated with it, which is the largest packet size (in octets or bytes) which can be transmitted or received on the interface. This configuration variable can be used to specify the desired MTU for the interface.

It is important that for given network link or media, that each host agree on a common MTU to be used. If no MTU is explicitly specified, then other encapsulators (such as the SLIP encapsulator) may supply a default value.

**Config(pni:ADDRESS)** Configuration Variable

When the pseudo network interface is enabled and configured, this parameter specifies what the local IP address of the interface will be. It should be expressed in the decimal digits dotted-quad notation, such as '**10.1.0.17**' or '**192.5.214.1**'.

**Config(pni:REMOTEADDRESS)** Configuration Variable

Normally the pseudo network interface is used as a point-to-point network interface. The **REMOTEADDRESS** parameter specifies the IP address of the host at the *other* end of the point-to-point interface. It is also specified in the decimal digits dotted-quad notation.

**Config(pni:NETWORKMASK)** Configuration Variable

Along with the **Config(pni:INTERFACE)** and **Config(pni:REMOTEADDRESS)** variables, this is used to specify the subnet mask associated with the local address of the network interface.

**Config(pni:DEFAULT)** Configuration Variable

This configuration variable takes an integer value. If the value of non-zero, then a default route pointing out this interface is installed. The default route will also be removed when the **pnid** daemon exits. This is only performed if the interface is a point-to-point network interface, like a SLIP serial line.

**Config(PNI:NOLOOPBACK)** Configuration Variable

If the network interface is a point-to-point network interface (it has a *REMOTEADDRESS* defined), then the usual case is to have the PNI encapsulator install a host route for the local IP address via the loopback interface. It will literally perform:

`/usr/etc/route add` *local-address* `127.0.0.1 0`

When the PNI interface is initialized. To *inhibit* this, define a value for Config(pni:NOLOOPBACK).

## 5.2 Configuration Primitives

*This section still under construction. . .*

# 6 FILTER Packet Filter Encapsulator

The **FILTER** encapsulator's purpose is to examine the packet which are being received and transmitted, and to take some action when the packet matches one or more of a set of filters. There are separate filters which are evaluated for inbound packets (*i.e.* those packets being decapsulated as they arrive from the network bound for the local host) as well as outbound packets (*i.e.* those packets sourced from the local host being transmitted to the network).

The actions which can be taken fall into one of a the following categories:

- Packets which are to be filtered out, or dropped if they appear.
- Packets which are used to reset the inactivity timer, if one exists, for this interface. Note that since there separate filters for inbound and outbound packets, you can have different inactivity times for each direction of packet travel.
- Packets which are *interesting* in some way, will will cause a one line log message to be written. For example. you might want to log the initial and final packets associated with every TCP connection which traverses the network interface associated with this encapsulator. The message logged is of the form:

```
log-input: 192.48.96.9 => 144.202.1.1 len 40 pr TCP port 20 -> 2704 SYN
```

The packet filters are specified in the form of symbolic packet filter expressions see Section 6.3 [Packet Filter Expressions], page 36. This is a very powerful and general mechanism which can specify the types of packets which you desire to select.

## 6.1 Configuration Directives

**Config(filter:INPUTFILTER)** Configuration Variable

The value of this configuration variable is a packet filter expression see Section 6.3 [Packet Filter Expressions], page 36 which defines which *input* packets are to be *excluded* and dropped on input.

**Config(filter:INPUTFILTER_RPCS)** Configuration Variable

This configuration variable, if present, should contain a list of Sun RPC program names. Any *input* packets directed to any of the specified RPC services will be dropped on input.

Examples of RPC program names are '**nfs**', '**ypserv**' and '**netinfod**'. See **man rpcinfo** for more details.

**Config(filter:OUTPUTFILTER)** Configuration Variable

The value of this configuration variable is a packet filter expression (see Section 6.3 [Packet Filter Expressions], page 36) which defines which *output* packets are to be *excluded* and dropped when transmitted.

**Config(filter:OUTPUTFILTER_RPCS)** Configuration Variable

**Config(filter:INPUT_LOG)** Configuration Variable

The value of this configuration variable is a packet filter expression (see Section 6.3 [Packet Filter Expressions], page 36) which defines which *input* packets should cause a log entry to be made.

**Config(filter:OUTPUT_LOG)** Configuration Variable

The value of this configuration variable is a packet filter expression (see Section 6.3 [Packet Filter Expressions], page 36) which defines which *output* packets should cause a log entry to be made.

**Config(filter:INPUT_ACT_INTERVAL)** Configuration Variable

Specify input inactivity timeout period. If no input packets arrive within the specified number of seconds, then the link will be taken down.

**Config(filter:INPUT_ACT_FILTER)** Configuration Variable

A packet filter expression which specified which *input* packets will cause the inactivity timer to be reset. If no filter is specified, then *any* input packet will reset the input inactivity timer.

**Config(filter:INPUT_ACT_ACTION)** Configuration Variable

If specified, a set of TCL commands which will be evaluated when the input inactivity period expires. If not specified, the default action will be to sever the connection.

**Config(filter:OUTPUT_ACT_INTERVAL)** Configuration Variable

Specify output inactivity timeout period. If no packets are transmitted within the specified number of seconds, the link will be taken down.

**Config(filter:OUTPUT_ACT_FILTER)** Configuration Variable

A packet filter expression which specified which *output* packets will cause the inactivity timer to be reset. If no filter is specified, then *any* output packet will serve to reset the output inactivity timer.

**Config(filter:OUTPUT_ACT_ACTION)** Configuration Variable

If specified, a set of TCL commands which will be evaluated when the output inactivity period expires. If not specified, the default action will be to sever the connection.

**Config(filter:DEMAND_FILTER)** Configuration Variable

*(Not yet fully functional).* The value of this configuration variable is a packet filter expression (see Section 6.3 [Packet Filter Expressions], page 36) which defines which *output* packets will cause a link level connection to actually be established, rather than immediately establishing a connection when the **pnid** process is started. This is used to implement “dial-on-demand”

## 6.2 Configuration Primitives

## 6.3 Packet Filter Expressions

The packet filter expressions used by the **FILTER** encapsulator are based on those used in the **tcpdump** program. The symbolic packet filter expression is compiled into a program which is evaluated by the Section 6.3.3 [Filter Machine], page 40, which is a simple state machine with a few registers and scratch memory store.

### 6.3.1 Packet Filter Syntax

[1]

The *expression* consists of one or more *primitives*. Primitives usually consist of an *id* (name or number) preceded by one or more qualifiers. There are three different kinds of qualifier: *type qualifiers* say what kind of thing the id name or number refers to. Possible types are **host**, **net** and **port**. E.g., **host foo**, **net 128.3**, **port 20**. If there is no type qualifier, **host** is assumed.

*dir* qualifiers specify a particular transfer direction to and/or from **id**. Possible directions are **src**, **dst**, **src or dst**, and **src and dst**. E.g., **src foo**, **dst net 128.3**, **src or dst port ftp-data**. If there is no dir qualifier, **src or dst** is assumed.

*proto* qualifiers restrict the match to a particular protocol. Possible protos are: **ip**, **tcp** and **udp**. E.g., **ip net 128.3**, **tcp port 21**. If there is no proto qualifier, all protocols consistent with the type are assumed. E.g., **src foo** means **ip src foo**, **net bar** means **ip net bar** and **port 53** means **(tcp or udp) port 53**.

In addition to the above, there are some special 'primitive' keywords that don't follow the pattern: **less**, **greater** and arithmetic expressions. All of these are described below.

More complex filter expressions are built up by using the words **and**, **or** and **not** to combine primitives. E.g., **host foo and not port ftp and not port ftp-data**. To save typing, identical qualifier lists can be omitted. E.g., **tcp dst port ftp or ftp-data or domain** is exactly the same as **tcp dst port ftp or tcp dst port ftp-data or tcp dst port domain**.

Allowable primitives are:

**dst host** '**host**'
: True if the IP destination field of the packet is '**host**'.

**src host** '**host**'
: True if the IP source field of the packet is '**host**'.

**host** '**host**'
: True if either the IP source or destination of the packet is '**host**'. Any of the above host expressions can be prepended with the keywords, **ip**.

**dst net** '**net**'
: True if the IP destination address of the packet has a network number of '**net**', which must be an must be an address.

[1] This section is an excerpt of the tcpdump(1) man page. The **tcpdump** program and man page was written by Van Jacobson (van@helios.ee.lbl.gov), Craig Leres (leres@helios.ee.lbl.gov) and Steven McCanne (mccanne@helios.ee.lbl.gov), all of Lawrence Berkeley Laboratory, University of California, Berkeley, CA.

**src net** '**net**'
: True if the IP source address of the packet has a network number of '**net**'

**net** '**net**'
: True if either the IP source or destination address of the packet has a network number of '**net**'.

**dst port** '**port**'
: True if the packet is ip/tcp or ip/udp and has a destination port value of '**port**'. The '**port**' can be a number or a name used in /etc/services (see tcp (4P) and udp (4P)).

  If a name is used, both the port number and protocol are checked. If a number or ambiguous name is used, only the port number is checked (e.g., **dst port 513** will select both tcp/login traffic and udp/who traffic, and **port domain** will select both tcp/domain and udp/domain traffic).

**src port** '**port**'
: True if the packet has a source port value of '**port**'.

**port** '**port**'
: True if either the source or destination port of the packet is '**port**'. Any of the above port expressions can be prepended with the keywords, **tcp** or **udp**, as in:

```
tcp src port `port'
```

  which matches only tcp packets.

**less** '**length**'
: True if the packet has a length less than or equal to '**length**'. This is equivalent to:

```
len <= `length'
```

**greater** '**length**'
: True if the packet has a length greater than or equal to '**length**'. This is equivalent to:

```
len >= `length'
```

**ip proto** '**protocol**'
: True if the packet is an ip packet (see ip (4P)) of protocol type '**protocol**'. '**Protocol**' can be a number or one of the names **icmp**, **udp**, **nd**, or **tcp**. Note that the identifiers **icmp**, **udp**, and **tcp** are also keywords and must be escaped via backslash ****, which is **** in the C-shell.

**ip broadcast**
: True if the packet is an IP broadcast packet. It checks for both the all-zeroes and all-ones broadcast conventions, and looks up the local subnet mask.

**ip multicast**
: True if the packet is an IP multicast packet.

**tcp, udp, icmp**
: Abbreviations for:

```
ip proto `Ip'
```

  where '**Ip**' is one of the above protocols.

**expr relop expr**

True if the relation holds, where **relop** is one of **>**, **<**, **>=**, **<=**, =, !=, and '**expr**' is an arithmetic expression composed of integer constants (expressed in standard C syntax), the normal binary operators [**+**, -, *, /, &, **|**], a length operator, and special packet data accessors. To access data inside the packet, use the following syntax:

```
proto [ `expr' : `size' ]
```

Proto is one of **ip, tcp, udp**, or **icmp**, and indicates the protocol layer for the index operation. The byte offset, relative to the indicated protocol layer, is given by '**expr**'. '**Size**' is optional and indicates the number of bytes in the field of interest; it can be either one, two, or four, and defaults to one.

The length operator, indicated by the keyword **len**, gives the length of the packet.

The expression **ip[0] & 0xf != 5** catches all IP packets with options. The expression **ip[2:2] & 0x1fff = 0** catches only unfragmented datagrams and fragment zero of fragmented datagrams. This check is implicitly applied to the **tcp** and **udp** index operations. For instance, **tcp[0]** always means the first byte of the TCP header, and never means the first byte of an intervening fragment.

Primitives may be combined using:

- A parenthesized group of primitives and operators (parentheses are special to the Shell and must be escaped).
- Negation (**!** or **not**).
- Concatenation (**and**).
- Alternation (**or**).

Negation has highest precedence. Alternation and concatenation have equal precedence and associate left to right. Note that explicit **and** tokens, not juxtaposition, are required for concatenation.

If an identifier is given without a keyword, the most recent keyword is assumed. For example,

```
not host vs and ace
```

is short for

```
not host vs and host ace
```

which should not be confused with

```
not ( host vs or ace )
```

*BPF EXAMPLES*

**tcp[13] & 3 != 0**

To select the start and end packets (the SYN and FIN packets) of each TCP conversation.

**ip[2:2] > 576**

To select IP packets longer than 576 bytes.

**ip[16] >= 224**

To select IP broadcast or multicast packets

**icmp[0] != 8 and icmp[0] != 0**

To select all ICMP packets that are not echo requests/replies (i.e., not ping packets).

### 6.3.2 Packet Filter Expression Test Program

The **testfilt** program, available as '**/etc/pni/FILTER.encap/testfilt**' can be used to test the syntax of a packet filter expression as it is being developed for use in the PNI package. It is not used at all in the normal operation of the package, and exists solely as a debugging and development aid.

The **testfilt** program takes as arguments the packet filter expression. Note that to ensure that expression are correctly evaluated and not interpreted by the shell, the expression should be enclosed in single quote characters. The **testfilt** program will then display the compiled BPF program in symbolic form, and will then attempt to evaluate the expression against a built in sample packet.

The sample packet is a TCP segment from host 1.1.1.2 to 1.1.1.2, from port 23 to port 4006 with a RST bit set in the flags field. Here is the exact contents of the sample packet:

```
static unsigned char pkt[] = {
    0x45, 0x00, 0x00, 0x28,
    0x21, 0x6c, 0x00, 0x00,
    0x3c, 0x06, 0x59, 0x5f,
    0x01, 0x01, 0x01, 0x02, /* src addr */
    0x01, 0x01, 0x01, 0x02, /* dst addr */
    0x00, 0x17, 0x0f, 0xa6,     /* src port, dst port */
    0x77, 0xa8, 0x42, 0xc1,     /* seq */
    0x00, 0x00, 0x00, 0x00,     /* ack */
    0x50, 0x04, 0x00, 0x00,     /* flags = RST, win = 0 */
    0xe1, 0xb4, 0x00, 0x00      /* cksum, urgent pointer */
};
```

This sample packet is then evaluated against the compiled packet filter program a large number of times to determine the length of time required to perform the evaluation.

Here is a sample invocation of the **testfilt** program:

```
$ /etc/pni/FILTER.encap/testfilt 'not (src host 111.11.128.10 or
src host 111.11.253.10 or src host 111.11.254.10) and
(udp dst port 161 or tcp dst port login or tcp dst port shell)'
worked
(000) ld       #0x0
(001) ld       [12]
(002) jeq      #0x6f0b800a      jt 20    jf 3
(003) jeq      #0x6f0bfd0a      jt 20    jf 4
(004) jeq      #0x6f0bfe0a      jt 20    jf 5
(005) ldb      [9]
(006) jeq      #0x11            jt 7     jf 12
(007) ldh      [6]
(008) jset     #0x1fff          jt 20    jf 9
(009) ldxb     4*([0]&0xf)
(010) ldh      [x + 2]
(011) jeq      #0xa1            jt 19    jf 20
(012) jeq      #0x6             jt 13    jf 20
(013) ldh      [6]
(014) jset     #0x1fff          jt 20    jf 15
(015) ldxb     4*([0]&0xf)
(016) ldh      [x + 2]
```

```
(017) jeq      #0x201           jt 19   jf 18
(018) jeq      #0x202           jt 19   jf 20
(019) ret      #149
(020) ret      #0

Timing 100000 invocations of filter..
Filter:   25.83456 microsecs per eval
```

To interpret the generated packet filter program, refer to the following section, Section 6.3.3 [Filter Machine], page 40. The time reported by **testfilt** only applies to the time taken to evaluate against the "test" packet; it may take more or less time in actual use depending on the types of packets coming across the link and the complexity of the packet filter expression.

### 6.3.3 Filter Machine

A filter program is an array of instructions, with all branches forwardly directed, terminated by a **return** instruction. Each instruction performs some action on the pseudo-machine state, which consists of an accumulator, index register, scratch memory store, and implicit program counter.[2]

The material presented in this section is background information, and is of assistance when examining the output of the **testfilt** program described above. It is not necessary (or even possible) to specify packet filter programs in this "raw" format.

The following structure defines the instruction format:

```
struct bpf_insn {
u_short code;
u_char  jt;
u_char  jf;
long k;
};
```

The *k* field is used in different ways by different instructions, and the *jt* and *jf* fields are used as offsets by the branch instructions. The opcodes are encoded in a semi-hierarchical fashion. There are eight classes of instructions: **BPF_LD**, **BPF_LDX**, **BPF_ST**, **BPF_STX**, **BPF_ALU**, **BPF_JMP**, **BPF_RET**, and **BPF_MISC**. Various other mode and operator bits are or'd into the class to give the actual instructions.

Below are the semantics for each defined BPF instruction. We use the convention that A is the accumulator, X is the index register, P[] packet data, and M[] scratch memory store. P[i:n] gives the data at byte offset "i" in the packet, interpreted as a word (n=4), unsigned halfword (n=2), or unsigned byte (n=1). M[i] gives the i'th word in the scratch memory store, which is only addressed in word units. The memory store is indexed from 0 to BPF_MEMWORDS-1. *k*, *jt*, and *jf* are the corresponding fields in the instruction definition. "len" refers to the length of the packet.

**BPF_LD** These instructions copy a value into the accumulator. The type of the source operand is specified by an "addressing mode" and can be a constant (**BPF_IMM**), packet data

[2] This section is excerpted from the bpf(4) manual page which described the Berkeley Packet Filter. The Berkeley Packet Filter was originally implemented by Steven McCanne of Lawrence Berkeley Laboratory. Much of the design is due to Van Jacobson.

at a fixed offset (**BPF_ABS**), packet data at a variable offset (**BPF_IND**), the packet length (**BPF_LEN**), or a word in the scratch memory store (**BPF_MEM**). For **BPF_IND** and **BPF_ABS**, the data size must be specified as a word (**BPF_W**), halfword (**BPF_H**), or byte (**BPF_B**). The semantics of all the recognized BPF_LD instructions follow.

**BPF_LD+BPF_W+BPF_ABS**
A <- P[k:4]

**BPF_LD+BPF_H+BPF_ABS**
A <- P[k:2]

**BPF_LD+BPF_B+BPF_ABS**
A <- P[k:1]

**BPF_LD+BPF_W+BPF_IND**
A <- P[X+k:4]

**BPF_LD+BPF_H+BPF_IND**
A <- P[X+k:2]

**BPF_LD+BPF_B+BPF_IND**
A <- P[X+k:1]

**BPF_LD+BPF_W+BPF_LEN**
A <- len

**BPF_LD+BPF_IMM**
A <- k

**BPF_LD+BPF_MEM**
A <- M[k]

**BPF_LDX** These instructions load a value into the index register. Note that the addressing modes are more restricted than those of the accumulator loads, but they include **BPF_MSH**, a hack for efficiently loading the IP header length.

**BPF_LDX+BPF_W+BPF_IMM**
X <- k

**BPF_LDX+BPF_W+BPF_MEM**
X <- M[k]

**BPF_LDX+BPF_W+BPF_LEN**
X <- len

**BPF_LDX+BPF_B+BPF_MSH**
X <- 4*(P[k:1]&0xf)

**BPF_ST** This instruction stores the accumulator into the scratch memory. We do not need an addressing mode since there is only one possibility for the destination.

**BPF_ST** M[k] <- A

**BPF_STX** This instruction stores the index register in the scratch memory store.

**BPF_STX** M[k] <- X

**BPF_ALU** The alu instructions perform operations between the accumulator and index register or constant, and store the result back in the accumulator. For binary operations, a source mode is required **BPF_K** or **BPF_X**).

**BPF_ALU+BPF_ADD+BPF_K**
A <- A + k

**BPF_ALU+BPF_SUB+BPF_K**
A <- A - k

**BPF_ALU+BPF_MUL+BPF_K**
A <- A * k

**BPF_ALU+BPF_DIV+BPF_K**
A <- A / k

**BPF_ALU+BPF_AND+BPF_K**
A <- A & k

**BPF_ALU+BPF_OR+BPF_K**
A <- A | k

**BPF_ALU+BPF_LSH+BPF_K**
A <- A << k

**BPF_ALU+BPF_RSH+BPF_K**
A <- A >> k

**BPF_ALU+BPF_ADD+BPF_X**
A <- A + X

**BPF_ALU+BPF_SUB+BPF_X**
A <- A - X

**BPF_ALU+BPF_MUL+BPF_X**
A <- A * X

**BPF_ALU+BPF_DIV+BPF_X**
A <- A / X

**BPF_ALU+BPF_AND+BPF_X**
A <- A & X

**BPF_ALU+BPF_OR+BPF_X**
A <- A | X

**BPF_ALU+BPF_LSH+BPF_X**
A <- A << X

**BPF_ALU+BPF_RSH+BPF_X**
A <- A >> X

**BPF_ALU+BPF_NEG**
A <- -A

**BPF_JMP** The jump instructions alter flow of control. Conditional jumps compare the accumulator against a constant (**BPF_K**) or the index register (**BPF_X**). If the result is true (or non-zero), the true branch is taken, otherwise the false branch is taken. Jump offsets are encoded in 8 bits so the longest jump is 256 instructions. However, the jump always (**BPF_JA**) opcode uses the 32 bit **k** field as the offset, allowing arbitrarily distant destinations. All conditionals use unsigned comparison conventions.

**BPF_JMP+BPF_JA**
pc += k

**BPF_JMP+BPF_JGT+BPF_K**
pc += (A > k) ? jt : jf

**BPF_JMP+BPF_JGE+BPF_K**
pc += (A >= k) ? jt : jf

**BPF_JMP+BPF_JEQ+BPF_K**
pc += (A == k) ? jt : jf

**BPF_JMP+BPF_JSET+BPF_K**
pc += (A & k) ? jt : jf

**BPF_JMP+BPF_JGT+BPF_X**
pc += (A > X) ? jt : jf

**BPF_JMP+BPF_JGE+BPF_X**
pc += (A >= X) ? jt : jf

**BPF_JMP+BPF_JEQ+BPF_X**
pc += (A == X) ? jt : jf

**BPF_JMP+BPF_JSET+BPF_X**
pc += (A & X) ? jt : jf

**BPF_RET** The return instructions terminate the filter program and specify the amount of packet to accept (i.e., they return the truncation amount). A return value of zero indicates that the packet should be ignored. The return value is either a constant (**BPF_K**) or the accumulator (**BPF_A**).

**BPF_RET+BPF_A**
accept A bytes

**BPF_RET+BPF_K**
accept k bytes

**BPF_MISC** The miscellaneous category was created for anything that doesn't fit into the above classes, and for any new instructions that might need to be added. Currently, these are the register transfer instructions that copy the index register to the accumulator or vice versa.

**BPF_MISC+BPF_TAX**
X <- A

**BPF_MISC+BPF_TXA**
A <- X

The BPF interface provides the following macros to facilitate array initializers:

```
  BPF_STMT(opcode, operand)
```
and
```
  BPF_JUMP(opcode, operand, true_offset, false_offset)
```

# 7 `SLIP` SLIP/CSLIP Encapsulator

The SLIP encapsulator implements the Serial Line IP encapsulation over a character stream, such as on a serial line. It uses a trivial encapsulation to denote boundaries between packets, and a simple byte-stuffing transparency scheme to escape the framing characters when they appear. SLIP is described in *RFC-1055, A NONSTANDARD FOR TRANSMISSION OF IP DATAGRAMS OVER SERIAL LINES: SLIP* by John Romkey.

The SLIP encapsulator also implements Van Jacobson TCP Header Compression, as documented in *RFC-1144, Compressing TCP/IP Headers for Low-Speed Serial Links* by Van Jacobson.

Use of the SLIP and CSLIP encapsulators is controlled by the specific license key string that was used to enable the software when it was installed.

If no MTU is specified as part of the configuration for the Chapter 5 [PNI Encapsulator], page 32, then the SLIP encapsulator will supply a default. If plain SLIP encapsulation is being used, the default MTU is 1006 bytes. If CSLIP encapsulation on the link is used instead, the default MTU is 256 bytes. The type of encapsulation used is specified by the *Config(slip:MODE)* configuration variable, described below.

## 7.1 Configuration Directives

**Config(slip:MODE)** Configuration Variable

This configuration variable varies the operation of the SLIP encapsulator. By default, it operations in plain SLIP mode. Setting the configuration variable to '`SLIP`' will result in the same effect.

Setting this variable to a value of '`CSLIP`' will enable the use of the Van Jacobson TCP Header Compression algorithm on this link. This results in less header overhead for TCP connections. Note that this is logically a *different* encapsulation than plain SLIP, so both end of the connection must agree on the encapsulation they are to use.

## 7.2 Configuration Primitives

*This section still under construction. . .*

# 8 **TTY** Serial Device Encapsulator

The **TTY** encapsulator sinks and sources streams of characters via serial tty-like device running in **RAW** mode (see the *tty(4)* UNIX manual page for a description of the various TTY modes). The **TTY** encapsulator open a serial device, usually '**/dev/cufa**' or '**/dev/cufb**' for outgoing connections, and will use the tty-device attached to file descriptor 0 ('**standard-input**') when used in "network access server" mode.

Since any tty device that implements the usual **ioctl** commands such as **TIOCSETP** is compatible with the **TTY** encapsulator, non-traditional serial devices such as '**TTYDSP**' or SCSI attached serial port expansion products can also be used. The PNI software can also be used in network access server mode over incoming telnet or rlogin connections (which use UNIX pseudo-tty devices), unlike TransSys DialUp-IP.

## 8.1 Configuration Directives

**Config(tty:DEVICE)** Configuration Variable

The value of this configuration variable should contain the name of the device to be used for outgoing connections (*e.g.*, '**/dev/cufa**').

This value need not be set for configurations used in network access server (incoming) connections.

**Config(tty:LOCKINGPROTO)** Configuration Variable

This configuration variable specifies what *locking protocol* is used to arbitrate access to the serial device. Serial devices are commonly shared amongst different programs such as '**uucp**', '**kermit**', '**cu**' and '**tip**' as well as other subsystems such as FAX modem drivers like '**NXFax**'. By observing the locking protocol, more than one program will not attempt the use of a device while it is in use by another user or another program.

There are two locking protocols supports by PNI: The '**UUCP Locking Protocol**' which is denoted by the **UUCPLOCK** keyword, which creates lock files in the '**/usr/spool/uucp/LCK**' directory. The other is no locking protocol at all, denoted by the **NONE** keyword. In the latter case, **pnid** will not attempt any locking at all before using the device named in the **Config(tty:DEVICE)** variable.

**Config(tty:DEVOPTS)** Configuration Variable

This configuration parameter is used to specify other device options to be enabled or disabled by the TTY encapsulator. The values of this configuration variable should be a TCL list of one or more of the following options:

Currently, on a single option is enabled:

IGNOREDCD

Ignore the state of DCD or Carrier Detect. Normally, the TTY encapsulator will end the current session if it discovers that the DCD has been lost on the serial connection. Specifying IGNOREDCD in the list of options will cause the DCD state to be ignored; this may be useful if there is no way to provide the correct modem control signals to the serial port.

**Config(tty:SPEED)** Configuration Variable

This configuration parameter is used to specify the serial port speed to be used for outgoing connections. It takes an integer value from the set of serial device speeds documented in the *tty(4)* UNIX manual page.

The supported speeds are: '**50**', '**75**', '**110**', '**134**', '**150**', '**200**', '**300**', '**1200**', '**2400**', '**4800**', '**9600**', '**19200**' and '**38400**'. On NEXTSTEP 3.0 and later systems some additional speeds are available: '**14400**', '**28800**', '**43200**', and '**57600**'. Note that while it may be possible to set the speed of the device at one of the higher speeds, the actual system may not be able to service the data arriving fast enough without dropping characters.

Experience has shown that a 25MHz 68040 CPU in a NeXTStation or NeXT Cube can comfortably support a serial port at 38400 bits per second.

For users of TTYDSP, the **TTY** encapsulator recognizes a TTYDSP serial port and will allow direct specification of *any* speed support by the TTYDSP product, rather than just the standard UNIX serial speeds.

**Config(tty:DIALER)** Configuration Variable

The DIALER configuration variable is used to specify which *Dialer* object is to be used by the **TTY** encapsulator to dial the modem and log into remote network access servers. The Dialer object implements the TCL-based scripting language to implement the chat scripts that talk to the modem and remote systems.

Currently, there is only one Dialer implementation available which is called *Dialer*. The class is dynamically loaded from the '**/etc/pni/Dialer.bundle**' directory on demand. This Dialer is used if no other one is specified.

**Config(tty:MODEMTYPE)** Configuration Variable

This variable is used to specify the type of modem which is being used for an outgoing connection. It is assumed to be attached to the device specified above. The modem type is used to select a file containing a dialing script to initialize, control and dial the modem to connect to a modem at a specified telephone number.

If, for example, the modem is a ZyXEL modem, the configuration would be something like:

```
set Config(tty:MODEMTYPE)   zyxel
```

which would cause the script in the file '**/etc/pni/support/dial-zyxel.tcl**' to be used. The modem dialing file (*e.g.*, '**dial-zyxel.tcl**') will be searched for in a number of directories in this order: '**/etc/pni/config**', '**/etc/pni/support**', and in the '**/etc/pni/TTY.encap**' directory.

Standard supplied dialing scripts are:

**dial-digicom.tcl**
: For the Digicom Systems 9624LE V.32 external modem. *Not* a modem which is particularly recommended.

**`dial-multitech.tcl`**
: For the Multitech MT932EA V.32 modem.

**`dial-multitech1432.tcl`**
: For the Multitech MT1432EA V.32/V.32*bis* modem.

**`dial-null.tcl`**
: This dialing script assume no modem at all; that is a direct, hardwired connection exists between two systems. This script is null and does nothing at all.

**`dial-worldblazer.tcl`**
: This dialing script is for the Telebit Worldblazer modem, which is configures for V.32/V.32*bis* connections and *not* PEP connections which operate badly in SLIP and PPP applications.

**`dial-worldport.tcl`**
: This dialing script can be used with the US Robotics WorldPort pocket modem, configured for V.32/V.32*bis* connections, with hardware flow control.

**`dial-zyxel.tcl`**
: Dialing script for the ZyXEL U1496, U1496E and U1496E+ series of modems. Connections are made using V.32, V.32*bis*, or the ZyXEL proprietary 16800bps and 19200bps modulations. V.42*bis* is used for reliability and data compression. This modem is recommended and is also supported by NXFax.

### Config(tty:SERVERTYPE) — Configuration Variable

In a manner similar to the MODEMTYPE variable just described, this configuration variable specified the type of remote network access server which is being called for outgoing connections. This is used to select a file which contains TCL scripting commands which “log in” to the remote network access server system, and condition it to enter its SLIP or PPP mode of operation. As part of this process, it may negotiate a user-name and password dialog as well.

The following network access server login scripts exist in the ‘**`/etc/pni/support`**’ directory:

**`login-annex.tcl`**
: Script file for a Xylogics (formerly Encore) Annex terminal server. It can go through a user name and password login process, and will put the port into SLIP mode from the CLI command prompt after login.

**`login-cisco.tcl`**
: Script file to log into a Cisco terminal server and enter SLIP mode.

**`login-null.tcl`**
: Null script file which is used when the remote network access server is already in SLIP mode and doesn’t require any login or interactive command process.

**login-pni.tcl**
: Script file to be used when the remote system is another instance of the PNI package running in “network access server” mode.

**login-pniserver.tcl**
: Not normally specified by the user, but the script file that is used by **pnid** when operating in server mode.

**login-portmaster.tcl**
: Simple script file which has been used to log into a Livingston Portmaster server. It assumes that the account to be used is configured to operate in SLIP mode.

**login-unix.tcl**
: Generic login script for a remote UNIX system that handles login and password dialog. Good starting point, perhaps for other systems.

There may be additional dialer and login scripts included which are not listed here. Please check the ‘**/etc/pni/support**’ directory to see which are available.

**Config(tty:NUMBER)** Configuration Variable

Used to specify the telephone number to dial. More than one number may be listed in a TCL list:

```
set Config(tty:NUMBER)  555-1212
```

or for multiple numbers..

```
set Config(tty:NUMBER)  { 555-1212 555-2121 }
```

**Config(tty:SERVERNAME)** Configuration Variable

This variable specified the name of the remote server. This name only have local significance, and is used to select, among other thing, the remote user name and password associated with a particular remote server.

The reason for this level of indirection is that it is possible to configure a rather complex configuration where multiple devices (each with their own modem type) have multiple telephone numbers and multiple remote servers associated with them. Each remote server might have its own distinct username and password that is required to complete the connection. (This complex configuration is not currently described.)

**Config(tty:DIALSCRIPT)** Configuration Variable

Internal configuration variable computed from other configuration information.

**Config(tty:DEVICES)** Configuration Variable

Internal configuration variable computed from other configuration information.

**Config(tty:PHONENUMBERS)** Configuration Variable

Internal configuration variable computed from other configuration information.

## 8.2 Configuration Primitives

*This section still under construction. . .*

# 9 `Dialer` support module

The `Dialer` object is used to support some encapsulators, like the `TTY` encapsulator, which need to perform scripting actions to establish a link-level connection. This usually involves initializing a modem, dialing the remote system, navigating through a login/password sequence, etc. The `Dialer` object is (the only, so far) implementation of an object which can perform this scripting and interaction on behalf of an encapsulator. The `TTY` encapsulator, for instance, allow the specification of the a Dialer object class to be specified, though it will default to the supplied `Dialer` object.

The purpose of the Dialer object is to communicate and interact with some remote entity using serial streams of characters. The encapsulator which uses the Dialer object (the TTY encapsulator in the PNI package) is responsible for the actual sending and receiving of the character streams.

The interaction done by the `Dialer` object is controlled by a powerful procedural language implementation, based on TCL with extensions. This is much more powerful than the usual SEND/EXPECT interface provided by dialers in UUCP and other software.

When the Dialer object is instantiated (again, usually by the `TTY` encapsulator), a new TCL command is created in the "current" TCL interpreter; by default, it is called "Dialer". However, in the PNI package, the name of the command is passed into the dialing procedure as an argument, so it is not necessary to know the name of the new TCL command when creating a script file. The *Dialer* TCL command then has a number of sub-commands which are used to perform the interactions required.

**dialer** *break* — Dialer TCL subcommand

This will cause a 750 millisecond break to be transmitted. A break is not a character, but a line condition, where the serial line, which normally idles in the "marking" (or 1) state is pulled to the "space" (or 0) condition for a period of time that exceeds the longest time it takes a character to be transmitted.

**dialer sleep** *snooze-time* — Dialer TCL subcommand

This causes execution of the dialing script to pause for the specified interval (in seconds).

**dialer** *status* — Dialer TCL subcommand

This command will return a list which describes the current state of the communications line that the Dialer command is using. This state normally includes a list of which RS-232 modem control signals are asserted.

**dialer parity** *zero*|*one*|*even*|*odd* — Dialer TCL subcommand

This command can be used to set the parity of the data which is sent by the `xmit` or `send` command. By default, all strings are sent with 8 bit, no parity. That is, normal ASCII characters are transmitted with the high bit set to *zero*. Other possibilities are specifying *even* or *odd* parity, or *one* to cause the high bit to be set to one.

**dialer xmit** *string* — Dialer TCL subcommand

The xmit subcommand is used to transmit a character string as part as a dialog to dial a modem or log into a remote system. Each character in the string is transmitted with the current parity setting.

**dialer delay** *inter-char-delay* — Dialer TCL subcommand

On some occasions, it is necessary to transmit strings of characters to remote devices and modems, but with some delay between each character. This is usually the case with some modems that cannot accept a stream of characters at "full" speed, and cannot perform flow control while in command mode. The **delay** subcommand takes one parameter, which is the delay time (in milliseconds) to be inserted betwixt characters being transmitted.

**dialer expect** *pattern action..* — Dialer TCL subcommand

The **expect** dialer command is used to scan the input stream for patterns or strings of characters. The command takes a list of one or more pairs of arguments. The first TCL argument of each pair is the pattern to be matched, while the second of each pair is a TCL expression to be evaluated should the pattern match.

It is important to recall how TCL programs are scanned and broken up into list arguments. It may be necessary to quote or group strings of words in patterns or TCL expressions so that they are treated as one argument to the expect command.

If there is a "missing" final argument (that is, an odd number of arguments), then there is no action associated the the last pattern specified.

The expect command will attempt matching of the input stream, subject to a timeout interval. Should that interval expire, a *timeout* condition occurs. The interval is defined (and can be modified by setting) the TCL variable *timeout*. When an **expect** command is invoked, the value of a local variable named *timeout* will be used; else the value of the global variable *timeout* is used instead.

The expect command will attempt to match each pattern as each character or group of characters arrive from the input stream. The input stream and patterns need not match lines of input, but rather any sequence of characters. As each pattern matching attempt occurs, each pattern is tested in order; the first pattern that matches will stop processing of the expect command after the TCL command associated with the pattern is evaluated.

There is a special instance of a pattern, **timeout**, which "matches" a timeout condition. If a timeout condition occurs, and there is no timeout pattern specified in an expect command, then a TCL error is raised.

Patterns can be specified in either UNIX **ed** regular expressions or by using shell wildcard or *glob* patterns. By default, the **expect** command uses the shell wildcard-style patterns, whilst the **rexpect** command uses regular expressions. It is possible to use both types of patterns in an expect or rexpect command by prefixing the pattern with an option to override the default expression type. For example, a pattern can be prefixed with an argument **-re** to cause the following expression to be treated as a regular expression. A prefix argument of **-glob** can be used to cause the immediately following pattern to be interpreted as a shell wildcard or *glob* type pattern.

The TCL expression associated with the pattern can be a single command or a sequence of TCL statements separated by semi-colons. (In the examples that follow, *$DIALER* is the name of the Dialer object being used and which implements the **expect** sub-command.)

```
$DIALER expect "*login:*"   { send "$password\n" }
```

The text matched by a "glob" type pattern is available in the *expect_match* TCL variable. However, this is likely not of tremendous use as in the example above *all* of current input text would have been matched.

If the pattern is specified using regular expressions, then much finer control is available, and it is possible to fetch parts of the text being matched when using the grouping constructs in a regular expression. As an example:

```
$DIALER expect -re \
 {Your IP address is ([0-9]+\.[0-9]+\.[0-9]+\.[0-9]+)}
```

Will look for an IP address specified in the dotted-decimal notation to immediately follow the phrase *Your IP address is*. The text that matches the part of the pattern enclosed in the *()* grouping constructs is available in a TCL variable. If there are more than one set of balanced *( )* pairs, they are also available making it possible to match and capture multiple bit of information on each patter match.

An array, called *$DIALER_expect_out* is created which contains the members for each grouped patterned specified in the pattern, starting with 1. The array elements defined for the first pattern are:

*1,start*
: The starting character position of the grouped pattern in the string which was matched.

*1,end*
: The ending character position of the grouped pattern i the string which was matched.

*1,string*
: The substring associated with the grouped pattern in the string which was matched. This string could be referenced as **set foo $DIALER_expect_out(1,string)**

The string which match each of the grouped patterns are also available in the variables *$1*, *$2*, *$3*, etc.

When a pattern match succeeds, the complete string that was matched against (that is, the current input buffer) is available in the TCL variable *expect_match*. This is the string which the *start* and *end* indices described above index into.

Note that strings which were matched and captured in the variables described may have multiple embedded carriage return, line feed and other control characters. This can cause problems with the **syslog** command since the system logging daemon will truncate text at the first newline character. The **printify** TCL command can be used to protect against this problem.

## 9.1 Sample Dialer Scripts

Included in this section are two scripts which are used by the **Dialer** class. A sample modem dialer and remote network access server login script are presented with commentary to describe their operation.

### 9.1.1 Sample Modem Dialer Script

The script that follows is based on the '**/etc/pni/support/dial-zyxel.tcl**' modem dialer script. It should be substantially similar to the one included with the PNI distribution.

Each modem dialing script is composed of a TCL procedure definition. This procedure will be invoked at the appropriate time when a serial line has been made available and the link needs to be established. The name of the TCL procedure should be prefixed by the constant string **dial-** followed by the name of the modem type (*e.g.*, **zyxel**). The modem type name is the one specified in the configuration file when specifying the type of modem in use.

The modem dialing procedure will be invoked with exactly three arguments:

1. The name of the Dialer object in use, so that its subcommands (like **expect**) can be referenced.
2. The telephone number to be dialed.
3. The name to be used to reference configuration information in the **Config** array.

```
#
#  TCL script used to configure and dial a ZyXEL U-1496-type
#  V.32bis external modem.
#
proc dial-zyxel { DIALER number cfg} {
    #
    # This script assumes that the zero'th argument is the dialer object
    # to be used, and the next argument be number to be dialed.
    #
    global Config testMode

    syslog LOG_INFO "$DIALER: Start of ZyXEL dialing script, dialing $number"
```

Set the default dialing method (DTMF/Tone or pulse) to TONE.

```
    set how TONE
```

Now we attempt to get the attention of the modem to ensure that it is attached, turned on and functioning. The attention string ("AT") is transmitted to the modem, and we wait for it to respond with "OK". We'll make three attempts before giving up.

Note that the **timeout** action in the **expect** commands is null, which allows execution to continue to the next TCL command should a timeout condition occur. The timeout interval is set to 2 seconds.

The **foreach {once}** loop is an idiom which allows us to quickly exit the loop (which only occurs one time) by using the **break** TCL command.

```
# flush any pending command
$DIALER xmit {\r}
$DIALER sleep 1

# get modems attention
set timeout 2
foreach i {once} {
    $DIALER xmit {AT\r}
    $DIALER expect "{*OK\r*}" break timeout {}
    $DIALER xmit {AT\r}
    $DIALER expect "{*OK\r*}" break timeout {}
    $DIALER xmit {AT\r}
    $DIALER expect "{*OK\r*}" break timeout {}
    error "Could not get the modem's attention"
}
set timeout 5
```

Now we turn off the command echo by the modem by sending it an "ATE0" command and waiting for it to reply with a string containing "OK". Since this is no action specified for the **`*OK*`** pattern, execution just continues to the next TCL command when the OK response arrives.

If a timeout condition occurs, the script raises an error by using the TCL **`error`** command.

```
# turn off command echo
$DIALER xmit {ATE0\r}
$DIALER expect timeout {error "waiting for OK"} "*OK*"
```

This next sequence of code queries the modem for identifying information which usually includes the type of modem and the version of firmware running in the modem. On some occasions, it is useful to know what version firmware a modem has installed when debugging connections.

Note that in this code fragment, the **`rexpect`** command is used, which results in the patterns being interpreted as regular expressions rather than "glob" style patterns.

An "ATI" command is sent to the modem to query it for its type. If the response returned includes the strings "1496", which is a model number then we query the modem again with an "ATI1" command to get additional information.

Note the use of ( ) grouped patterns to isolate a part of the string being matched so that it may be referenced later on as **`$1`**.

```
$DIALER xmit {ATI\r}
set version ""
$DIALER rexpect timeout {} \
    "\n1496\r\n"        {
        $DIALER xmit {ATI1\r}
        $DIALER rexpect \
            timeout {} \
            "\[\r\n]+(.*).*OK\r\n" {set version $1}
    } \
    "\[\r\n]+(.*).*OK\r\n" {set version $1} \
    timeout    {error "Could not get modem version"}
```

Now we clean up the string that was matched and extracted, replacing all tab, carriage return and line feed characters with a single space using the TCL **regsub** command. We then squeeze out all occurrences of more than one blank to a single blank with the second **regsub** command. Finally, leading and trailing spaces are deleted using the last **regsub** command, and the message is logged.

```
if {[string length $version]>0} {
    regsub -all "(\t|\r|\n)+" $version " " version
    regsub -all " +" $version " " version
    regsub { +$} $version "" version
    syslog LOG_DEBUG "Modem firmware version is '$version'"
}

$DIALER sleep 1
```

We now prepare to configure the modem so that it can correctly pass SLIP traffic over the link. These parameters are, of course, specific to each type of modem and will surely be different for modems other than the ZyXEL 1496 series of modems.

```
# configure modem with proper parameters
# &K4        - V.42/V.42bis or MNP4/MNP5
# &N0        - auto-negotiate highest possible link rate
# M0        - speaker off
# V1        - verbose responses
# Q0        - display responses
# X5
# &C1        - CD tracks carrier presence
# &D3        - hang up and reset to profile 0 when DTR dropped
# &H3        - hardware (RTS/CTS) flow control
# &J0        - single phone line RJ11 jack
# &L0        - normal phone line (not leased)
# &M0        - async mode
# &R1        - ignore RTS, assume always on
# &S0        - DSR override, assume always on
# N1        - ring volume
# S2=128          - turn off escape into command mode
# S42.1=0         - disable throughput averaging
# S38.5=1         - disable use of MNP5 (only use V.42bis)
# S46.2=1         - disable pesky "RINGING" messages
# AT+FCLASS=0 - make sure the modem isn't in FAX mode
```

The **setupstr** variable is set to be a TCL list of parameters to be sent to the modem as described in the comments above. If we're in test mode (indicated by the **testMode** TCL variable existing), then append a modem command to enable the speaker whilst dialing the modem and until it connects.

```
set setupstr { +FCLASS=0 S42.1=0 S38.3=1 S38.5=1 S2=128 S46.2=1
    &K4 &N0 M0 V1 Q0 X5 &C1 &D3 &H3 &J0 &L0 &M0 &R1 &S0 N1 }
```

```
# turn on speaker during dialing and connect sequence if in test mode
if [info exists testMode] {
    lappend setupstr "M1"
}
```

Now, interate over each element of the list of modem commands specified above and sent it to the modem and await an OK response. Each command is done one at a time because some modem have problems will a long command string being sent to them in command mode. We can also tell which modem command cased an error by sending them one at a time.

```
foreach s $setupstr {
    $DIALER xmit "AT $s\r"
    $DIALER expect timeout {error "waiting for OK for parm $s"} \
        "{*ERROR\r*}" {warn "Modem returned ERROR setting $s"} \
        "{*OK*}"
}
```

Check the configuration array to see if a dialing type was specified or not.

```
# allow override of dialing type
if {[info exists Config($cfg:DIALTYPE)]} {
    set how $Config($cfg:DIALTYPE)
}
```

Now dial the phone. If the dialing type is neither “dtmf” or “tone”, the pulse dial using the modem’s ATDP command. Otherwise, dial using the modem’s ATDT command.

```
# dial the phone
if { [string compare [string tolower $how] tone] &&
     [string compare [string tolower $how] dtmf] } {
         notice "Dialing (pulse) $number ..."
         $DIALER xmit "ATDP$number\r"
     } else {
         notice "Dialing (DTMF) $number ..."
         $DIALER xmit "ATDT$number\r"
     }
```

We now wait for the modem to connect. This can take a rather long time for the since the call has to be made through the telephone system, and the modems train and connect. To accommodate that, we set the timeout for the **expect** command to be 45 seconds.

```
# wait for connect message
set timeout 45

set rng 0
```

We loop examining responses from the modem. These can include errors like "NO DIALTONE", "NO CARRIER" or "BUSY"; or progress messages such as "RINGING"; or a "CONNECT" message which we hope for.

```
    while 1 {
        $DIALER rexpect  \
          timeout   {
              error "Timeout waiting for modem to connect to $number "} \
          "(CONNECT.*)\r+\n"         {
              syslog LOG_INFO "Connected: $dialer_expect_out(1,string)"; \
              break \
          } \
          "\r\nERROR\r\n"      {error "Modem returned ERROR"} \
          "\r\nRINGING\r\n"    {syslog LOG_INFO "Ringing.. ($rng)"; incr rng} \
          "\r\nNO CARRIER\r"   {error "Modem returned NO CARRIER"} \
          "\r\nNO DIAL TONE\r" {error "Modem returned NO DIAL TONE"} \
          "\r\nBUSY\r\n"       {error "Remote modem busy"}
    }
}
```

### 9.1.2 Sample Network Access Server Login Script

This section examples a script with "logs in" to a network access server; in this particular instance it is a Telebit Netblazer. This file is derived from '**/etc/pni/support/login-netblazer.tcl**'.

It begins with a small "helper" TCL procedure which is called by the main procedure below.

```
proc nb-password { DIALER username password } {
    $DIALER xmit "$username\r"
    $DIALER expect "*assword:*"  {$DIALER xmit "$password\r"} \
        timeout  {error "waiting for password prompt"}
}
```

As was the case with the modem dialing scripts, the login scripts are expected to declare a TCL procedure named **login-***SystemType*, where SystemType is the type of remote network access server, and will be referenced in the configuration file's **Config** array.

The login procedure takes exactly two arguments, the first being the name of the Dialer object (as in the modem script), and the second being the name used to reference configuration information in the **Config** array.

To perform the login action, the remote network access server (in this case) requires a user name and password. This information is not embedded in the login script, but is stored elsewhere. It can be specified in the configuration file, but that may needlessly expose sensitive information. It is also possible to store that information in a separate file which can have restricted access (only to root, for instance) to protect it. This login script accommodate either approach and will in fact complain if the file which stores the username and password is insecure with inappropriate directory permissions.

```
# The username and password to be used are stored in a
# separate file.  The name of the file is specified in the
# Config array.
#
# This file should contain (at least) two TCL set commands to
```

```
# set the variables 'username' and 'password'.  Its likely
# that you don't want these files generally readable for
# security purposes, so a check is made for you, and the
# script will fail if the file is readable.
#
# Alternatively, the members of the keyed list "username" and
# "password" can be used to specify the username and password to be
# passed to the terminal server.

proc login-netblazer { DIALER cfg } {
    global Config

    syslog LOG_INFO "Begin netblazer login"

    set username ""
    set password ""

    if {[info exists Config($cfg:SECRETFILE)] &&
        [file exists $Config($cfg:SECRETFILE)]} {

            if {[file readable $Config($cfg:SECRETFILE)]} {
                source $Config($cfg:SECRETFILE)
            }

            file stat $Config($cfg:SECRETFILE) stat
            if {[expr $stat(mode)&04]} {
                error "File with password, $Config($cfg:SECRETFILE), \
is readable by 'other'!"
            }
        } else {
            if {[info exists Config($cfg:USERNAME)]} {
                set username $Config($cfg:USERNAME)
            }
            if {[info exists Config($cfg:PASSWORD)]} {
                set password $Config($cfg:PASSWORD)
            }
        }
```

Set the parity of the transmitted data to 8 bits, no parity. This isn't really necessary as this is the default state. Of course, the modem dialing script could have changed it, so we reset it here just to be safe.

A pair of carriage return characters are transmitted in case some auto-baud action needs to occur, and then we look for a login prompt from the network access server.

```
#
# set parity of transmitted data
#
$DIALER parity ZERO
#
#  For auto-baud nonsense.. sometimes its necessary to poke at it a couple
#  of times before it figures out what speed the modem is at.
#
set timeout 2
$DIALER xmit {\r\r}
$DIALER sleep 1

foreach i {once} {
    $DIALER xmit {\r}
    #
    #  look for host prompt
    #
    $DIALER expect timeout {} \
        "*login:*" {nb-password $DIALER $username $password; break}

    $DIALER xmit {\r}
    $DIALER expect timeout {} \
        "*login:*" {nb-password $DIALER $username $password; break}

    $DIALER xmit {\r}
    $DIALER expect timeout {} \
        "*login:*" {nb-password $DIALER $username $password; break}

    error "Couldn't autobaud login: prompt from NetBlazer"
}
set timeout 10
```

In the loop above, the nb-password procedure that was defined first is invoked to actually handle sending the username and password when a login prompt is seen.

This particular network access server is configured to automatically enter SLIP mode when the appropriate username and password is received, so there's no need to transmit a command to do so. We simply wait for a message indicating it is switching into packet mode and return.

```
#
#  send command to put terminal server into SLIP mode
#
set mode SLIP
if {[info exists Config($cfg:MODE)]} {
    set mode $Config($cfg:MODE)
}

$DIALER expect {*Packet\ mode\ enabled*} {} \
       timeout {error "Timeout waiting for Packet Mode message"}

syslog LOG_DEBUG "Entering Packet Mode"
```

```
    #
    # that's all
    #
    return "Connected"
}
```

# 10 **LOOP** Loopback Encapsulator

The **LOOP** encapsulator is a debugging aid which simply loops back any packets being encapsulated as input data. It can typically be used in place of a **TTY** (or even a combination of **SLIP** and **TTY**) encapsulator.

## 10.1 Configuration Directives

The **LOOP** encapsulator has no configuration directives.

## 10.2 Configuration Primitives

The **LOOP** encapsulator has no configuration primitives.

# 11 TUNNEL Virtual Path Tunnel Encapsulator

The **TUNNEL** encapsulator is used as an alternate means of transport in place of the **TTY** encapsulator. Rather then sending a stream of characters out a serial port by using a combination of the **SLIP** encapsulator followed by a **TTY** encapsulator, the TUNNEL encapsulator can be used. It will accept an input IP packet, and use UDP (*User Datagram Protocol*) to transport the packet to be sent across the internet. This capability make it possible to build a virtual network composed of virtual link over an existing IP infrastructure. That is, the tunnel encapsulator can be used to construct virtual "wires" which packets are transported over; it just so happens that the "wires" are carried over existing IP networks.

The **TUNNEL** encapsulator maintains a table which associates for each next-hop gateway or router address the tunnel endpoint IP address and UDP port number which the encapsulated packet is to be sent to.

Consider for a moment an Ethernet environment. When a packet needs to be transmitted, the network software determines the *next hop* for the packet. This is either the destination host's IP address if it is on the same ethernet or the next hop is the IP address of an intermediate router on the local ethernet. In the case of a point-to-point network interface, the "next hop" address is always the IP address of the host on the other end of the point-to-point link, since there are no other alternatives on that point-to-point network.

Let's take an example. Consider that we've defined an interface to be an tunnel endpoint. The interface's address is 10.1.0.1, with a subnet mask of 255.255.255.0. If we'd like to configure this as a point-to-point interface, we would need to have the address of the "other" side of the point to point link. Consider that it is 10.1.0.2 in this case.

So we have a point to point interface defined; the local address is 10.1.0.1 and the remote address is 10.1.0.2; so far this isn't much different than a normal serial interface. Here's where things start to get different: we define the virtual "wire" in the tunnel interface.

First we need to define the local host's end of the wire. The end of the wire is logically a UDP port on the local host. In this example we would choose port number 4342. Now we need to identify the other end of the virtual wire; it is going to terminal on another host on the Internet with the IP address of 26.0.0.73 on UDP port 2357, in this example. This means that there is *another* instance of the PNI software running on the remote system terminating the virtual wire.

The virtual wire is identified by the destination IP address associated with the network interface; in the case of a point-to-point interface, there can only be the "other" end. It is possible to configure a non point-to-point tunnel interface where there are multiple virtual wires, one to each other host on the virtual "network" which they all share. In the example scenario, the only virtual wire is identified by the remote host's IP address: 10.1.0.2. It would then be necessary to establish a mapping:

**10.1.0.2 ⇒ (ip addr 26.0.0.73, port 2357)**

conversely, there would be a similar mapping on the *remote* system which maps our address on the 10.1.0.0 subnet to our local ip address and port number.

How does the encapsulated packet to 10.1.0.2 get to the 26.0.0.73 host? Via some *other* functioning network interface on the host, following whatever route exists to that remote host. The tunnel encapsulator is the logical end of a string of encapsulators consisting usually of the PNI, (perhaps FILTER), and then TUNNEL.

## 11.1 Configuration Directives

**Config(tunnel:PORT)** Configuration Variable

This is the integer port number (from 1 to 65535) which the local tunnel encapsulator will bind and receive incoming packets on. The value of this selection must be coordinated with the remote host or hosts using tunnels.

**Config(tunnel:TUNNELS)** Configuration Variable

This configuration variable consists of a list of 3-tuples (each 3 element list) which map the next-hop IP address to a tunnel end-point IP address and UDP port number.

```
set Config(tunnel:TUNNELS) {
       { 10.1.2.3        144.202.0.1  1203 }
       { 10.1.3.3        26.0.0.73    1792 }
   }
```

In this example, we have configured two different tunnel end-points (either because this is a non-point-to-point network or because this part of the configuration file is share by both ends). Packets input to this **TUNNEL** encapsulator whose next hop IP address are 10.1.2.3 will be wrapped inside of UDP and sent over the Internet to the host 144.202.0.1, UDP port 1203 where, presumably, another PNI is running and waiting for packets to arrive. The remote host should also have an inverse mapping will will allow the return traffic to be sent back to this host to the UDP port number that we configured earlier.

## 11.2 Configuration Primitives

*This section still under construction. . .*

# 12 SECURE Data Security Encapsulator

## 12.1 Configuration Directives

**Config(secure:KEYS)** Configuration Variable

**Config(secure:HOSTS)** Configuration Variable

## 12.2 Configuration Primitives

*This section still under construction. . .*

## 12.3 Tamper-Proof Objective-C protocol definition

The **SECURE** encapsulator relies on a companion class to actually perform the verification algorithm on the data being transmitted and received. This companion class must conform to the **TamperProof** protocol described below.

```
/*
 * Copyright (c) 1993 by TransSys, Inc.
 * All rights reserved. This source code document is an unpublished
 * work and contains confidential and proprietary information.
 *
 */

/*
 * This file defines a protocol, TamperProof, which is used by the
 * SECURE encapsulator to verify that data which is transmitted
 * and received will not be tampered with in transit.
 */

@protocol TamperProof

/*
 * This first group of methods is used to discover what characteristics
 * the implementation of TamperProof has.
 *
 * First, define what the block size of the data to be checked is. It
 * is the responsibility of the encapsulator to provide data to be
 * tamperproofed in multiples of the block size.
 */
- (int) tamperBlockSize;
```

```
/*
 *  Some TamperProof algorithms may compute a result of of some sort; this
 *  method returns what the size of that result may be.  It is the
 *  responsibility of the caller to set aside space for this result, which
 *  is carried from end to end to aid in verification.
 */
- (int) tamperResultSize;

/*
 *  Specify keying data which is used to implement the TamperProof-ing
 *  algorithm.  This data is passed in as a pointer and length, and is
 *  otherwise opaque.  This same keying data is used for data being
 *  prepared for transmission, as well as checking arriving data to
 *  ensure it has not been tampered with.
 */
- tamperSetTamperKey:(void *)key length:(int) length;


/*
 *  Verify that the input data received is valid and has not
 *  been tampered with in transit.  This data will be evaluated in
 *  the context of the "tamper key" set in the method above, and with
 *  the signature data, if any, that was passed in.  This signature
 *  data is assumed to be of fixed length (returned by tamperResultSize).
 */
- (BOOL) tamperVerifyInput:(void *)data length:(int)length
                           with:(void *)signature;

/*
 *  Prepare data which is to be transmitted for tamper prevention or
 *  detection.  The computed output "signature", if any, will be returned
 *  and be of length specified by the tamperResultSize method.  The output
 *  data will be secured in the context of the key specified by
 *  tamperSetTamperKey: above.
 */

- tamperSecureOutput:(void *)data length:(int)length
                           to:(void *)signature;
@end
```

# 13 Operation

## 13.1 pnid

The **pnid** program is the central component of the PNI package. For more information on the operation of the **pnid** program, See Section A.4 [The **pnid** daemon], page 82.

The other major operational component of the PNI package, the **pni_reloc** kernel driver is not a normal UNIX program, but rather a loadable kernel driver, See Section A.3 [**pni_reloc** kernel driver], page 81.

### 13.1.1 Arguments and Options

**-d** This option is used for debugging. It configures the logging of the **pnid** program to be sent to standard error *only*. The usual logging is done by using the **syslog** facility. Also, this performs the **-C** option as well.

**-C** This option enables core-dump files if the **pnid** program terminates abnormally. Core dumps may not be produced by default if the '**ulimit**' command has disabled them.

**-n logtag**

This option takes a parameter, which is a string to be used to mark the logging messages produced by this instance of the **pnid** program. In the absence of this option, log messages look like this:

```
Oct 11 21:19:01 wa3ymh pnid[215]: TransSys PNI 1.1 ...
```

with a '**-n pni0**' option specified, the message would look like this instead:

```
Oct 11 21:19:01 wa3ymh pnid-pni0[215]: TransSys PNI 1.1 ...
```

**-f configfile**

This option takes a single argument which is the pathname of the configuration file to be loaded. If this option is not specified, then the file '**config**' in the current directory will be loaded.

**-o opt** Set the value of the *Option(opt)* variable to the value of **1**. Alternatively, if the syntax of '**opt**' is of the form '**opt=value**', then the **Option(opt)** variable is set to '**value**' instead of '**1**'.

The only option currently used is *Option(priority)*, which is used to vary the priority that the **pnid** process runs at. Note that to cause the daemon to run at a higher, preferred scheduling priority, a *negative* priority should be specified.

**-i** A debugging option. After loading the '**/etc/pni/support/init.tcl**' file and before load the configuration file, the **pnid** program enters an interactive mode. It will read and evaluate TCL expressions from standard input.

**-c** If specified, the **pnid** program will operate in '**client**' mode; the usual case when dialing out to a remote system. This has the effect of setting the TCL '**mode**' variable to the value **CLIENT**.

**-s** If specified, the **pnid** program will operate in '**network access server**' mode; the usual case when being connected to by a remote system. This has the effect of setting the TCL '**mode**' variable to the value **SERVER**.

**-e expression**
While parsing options and initializing, evaluate the '**expression**' argument as a TCL expression.

**-t** Debugging option. When specified, the **pnid** program operates in test mode. This has the effect of setting the TCL '**testMode**' variable to the value **1**. Otherwise, the '**testMode**' variable is left unset.

**-R keystring**
Register the program using the license key string specified. User is prompted on standard output for registration information (name, address, etc.), which is read from standard input.

### 13.1.2 Updated License Key Registration

The operation of the **pnid** program is controlled by a license key string supplied with the software. When the PNI software is installed, you will be asked if you would like to install it with a demonstration license key provided with the distribution or to specify one of your own.

The demonstration license key allows operation of various features for a limited time.

To re-register the PNI software with a different license key, it is necessary to run the **pnid** program as root. Perform the following commands, as root, from a UNIX shell window:

```
# mv /etc/pni/keyfile /etc/pni/keyfile.save
# /etc/pni/pnid -R 'supplied-key-string'
```

You will then be prompted for your name, address and other information which is stored on the local host in the file '**/etc/pni/keyfile**'. In many cases, it will not be possible to move this keyfile to another machine as its installation is tied to the particular hardware. This file is stored in a binary-encoded format and is not in a textual format. Do not attempt to modify the contents of that file, as it will render the installed keyfile invalid.

Should you need to move the software to another host, you can re-install it using the process described above.

### 13.1.3 pnid output and logging

The **pnid** daemon is normally started at boot time, and runs as a system daemon process. Since it is not interactive, when it needs to communicate with the system administrator or note some useful information, it will use *syslog* to do do, rather the writing messages to its standard output or standard error stream.

Dialing and login scripts can cause messages to be logged by using the **syslog** TCL command.

The syslog facility can route log messages based upon a *facility* and a *priority*. The facility can be used to identify a source of log messages or a particular class of messages. The **pnid** daemon uses the *local5* facility code when creating log messages; during installation (see Section 3.2.5.5 [Changes to /etc/syslog.conf], page 20) and entry is made in the syslog configuration file to direct these messages to the file '**/usr/adm/pni.log**' where they can be examined.

It is possible, however, to invoked the **pnid** program with the **-d** option to cause messages which would have otherwise been logged to be directed to the standard error stream instead (see Section 13.1.1 [Arguments and Options], page 65).

### 13.1.4 pnid termination conditions

Normally, the **pnid** process is relatively long-lived; it is started at boot time to establish a network connection and then continues to run until the system is halted or rebooted. Alternatively, the **pnid** process may be started "manually" either by a user directly, or from the cron program at a scheduled time.

The **pnid** process will normally run continuously. It is possible to configure it to exit after a period of network inactivity (see Chapter 6 [FILTER Packet Filter Encapsulator], page 34). The process will also exit if the physical connection in use becomes unavailable; *e.g.*, the modem hangs up due to loss of carrier from the remote.

To manually cause the **pnid** program to exit the **pnistat** script (see Section 13.3 [pnistat], page 67) can be used to issue an authenticated command to the **pnid** daemon to cause it to exit. A less elegant approach is to send the daemon process an interrupt (SIGINT) or terminate (SIGTERM) signal. A hangup signal (SIGHUP) will also cause the daemon to exit as if the carrier from the modem dropped.

## 13.2 The pnitcl program

The **pnitcl** program is an augmented, standalone TCL (Tool Control Language) interpreter used to build monitoring and control tools. It is an implementation of TCL as a standalone tool, rather than being embedded within another application program (such as within **pnid**). The **pnitcl** program can be used to execute standalone TCL scripts, in much the same way the the shell execute shell scripts.

There are three distinct set of commands available within the **pnitcl** program:

1. The standard set of core primitives available as part of the normal TCL package.
2. A set of extension primitives provided by the TCLX extension package. Note that a few of the extensions in TCLX have not been included which are not commonly used or otherwise might interfere with the use of TCL in the **pnid** program.
3. A set of extensions specific to the PNI package, including a primitive to perform RPC calls to instances of the **pnid** daemon.

## 13.3 The pnistat script

The **pnistat** script is used to query the operation of a running instance of the **pnid** daemon.

### 13.3.1 **pnistat** usage

The **pnistat** program is invoked from a UNIX shell to query and control a running instance of the **pnid** daemon. It is invoked as:

**pnistat 'options' 'pnidaemon_name'**

It takes the following options:

**-a** Run anonymously. This performs an unauthenticated query to the remote **pnid** daemon, rather than using a password from the '**/etc/pni/config/password**' file or from the **PNIPASSWORD** environment variable.

**-c commandname**
Perform the command, '**commandname**', in the remote **pnid** process. The default command is '**stat**' to obtain the status of the daemon process. The other useful command you might use is '**down**', to cause the **pnid** process to shutdown and exit.

**-d** Increment the debug level. Not normally used.

**-e encap** When using the '**stat**' command, this allow you to specify a particular encapsulator, '**encap**', to obtain information about rather than *all* encapsulators.

**-h host** Query the host '**host**' rather then the local host. Useful for monitoring the operation of a **pnid** daemon on a remote machine, assuming that there is network connectivity to the remote.

**-P password**
Specify an alternative password to perform an authenticated operation to the remote **pnid** daemon.

**-p pni***N* Add the pni daemon process, **pni***N*, to the list to be queried. The values specified are the symbolic names such as '**pni0**', '**pni1**', etc.

**-v** Set verbose mode. Not normally used. When verbose mode is enabled, the raw data returned by the daemon process is displayed.

### 13.3.2 **pnistat** *stat* command

The default output produced by the **pnistat** script is for the *stat* command which displays hopefully useful or interesting statistics about the **pnid** daemon. The output produced looks like this:

```
$ pnistat
                              Encapsulated       Decapsulated
   Type   Encap  state       Bufs      Bytes     Bufs     Bytes
   PNI*     pni  phase2     71417    5674790    69560  17260117
FILTER*  filter  phase2     71417    3389446    69560  17260117
  SLIP*    slip  phase2     71417    3389446   795123  15104232
   TTY*     tty  phase2     71418    1375574   795123  15104232
```

This format of output reports statistics gathered by each encapsulator in the running configuration.

The first column is the *type* of the encapsulator, whilst the second column is the *name* of a particular instance of an encapsulator. While not obviously useful in normal situations, it is possible to have more than one instance of a particular encapsulator in a single configuration.

The asterisk next to the encapsulator type indicates that the encapsulator is active; this will always be the case except when debugging certain situations during development.

The *state* column indicates the current state of the encapsulator; this should always be either '**phase1**' or '**phase2**' to indicate what stage of initialization the encapsulator has completed. The usual state is '**phase2**', which indicates that the associated encapsulator has been initialized completely and is up and ready to receive or transmit data.

The next two sets of columns count buffers of data and byte counts which are *encapsulated* and *decapsulated* at each layer. *Encapsulated* refers the process of moving the network traffic from the operating system out towards the network hardware. *Decapsulated* is the reverse; that is, taking streams of data from the network hardware and parsing them into frames and packets to be handed to the operating system network protocol stack.

More information is available when the **pnistat** script is invoked with the **-v** option:

```
$ pnistat -v
pni0: running since Fri Nov 26 00:07:21 1993 (22 hours, 9 minutes, 45 seconds)
                          Encapsulated      Decapsulated
   Type   Encap state      Bufs     Bytes    Bufs     Bytes
   PNI*     pni phase2   71765   5700150   69908  17356184
FILTER*  filter phase2   71765   3403670   69908  17356184
                       Idle time: 8 seconds
  SLIP*    slip phase2   71765   3403670  799580  15189576
        VJ TCP Hdr comp: 6415 uncomp/63459 comp   1380 uncomp/67025 comp
   TTY*     tty phase2   71765   1380568  799580  15189576
```

In this example, certain encapsulators report back statistics specific to their type. For instance, the '**FILTER**' encapsulator reports the length of time that traffic has been idle, since it enforces idle traffic timeouts. The '**SLIP**' encapsulator reports statistics specific to the Van Jacobson TCP Header Compression algorithm.

Also reported in the verbose format is the time that the **pnid** process was started (in UTC), and the time elapsed since the start time.

We note that this daemon has been pretty busy, with an almost constant stream of input data. The resources used by this daemon as reported by the **ps** command are:

```
$ ps xa|grep pnid
  335 ?  S    29:19 /etc/pni/pnid -C -c -n pni0 -f /etc/pni/config/pni0.config
10279 p2 S     0:00 grep pnid

$ ps v335
  PID TT STAT  TIME SL RE PAGEIN VSIZE RSIZE   LIM TSIZ TRS  %CPU %MEM COMMAND
  335 ?  S    29:20  0  0      0 2.11M  736K     0   0K   0   1.5  3.6 pnid

$ ps m335
USER      PID TT  %CPU STAT PRI    SYSTEM      USER  COMMAND
root      335 ?    6.1  S    20  21:17.56   8:03.04  /etc/pni/pnid -C -c -n pni0
```

```
$ ps 1335
       F  UID   PID  PPID  CP PRI BASE VSIZE RSIZE WCHAN STAT TT   TIME COMMAND
80000001    0   335   317  0  20   20 2.11M  736K     0  S    ?  29:23 /etc/pni/p
```

### 13.3.3 `pnistat` control operations - the *down* command

The *down* command is used to cause the **pnid** daemon to immediately exit. This will cause, in the usual case, the modem used for the connection to hang up and the physical connection be terminated.

The *down* command is used like this:

```
$ pnistat -c down pni0
```

This will cause the **pnid** daemon for the pni0 network interface to exit. In order to successfully perform the *down* command, you must perform an *authenticated* interaction with the **pnid** daemon. This requires that you know the proper password to authenticate the command. Normally, the proper password will be obtained from '**/etc/pni/config/password**' which is protected the UNIX file permissions so that only the *root* superuser has access as well as those users in the UNIX group *pni*:

```
$ ls -lg /etc/pni/config/password
-rw-r-----  1 root   pni   33 Sep 17 22:54 /etc/pni/config/password
```

## 13.4 The '`/etc/pni/pnirun`' shell script

The **pnirun** shell script is used to start (and stop) instances of the **pnid** daemon process. It is intended to be used in two different ways: at boot time, invoked from the '**/etc/rc.local**' file; and interactively, invoked by hand to bring up (or take down) a network connection as desired.

The **pnirun** script takes a number of options:

**-v**

**-x** These options set the corresponding '**/bin/sh**' options and are used for debugging purposes only.

**-all** Start instances of the **pnid** daemon process for all interfaces for which configuration files exist.

**-boot** Indicates that the **pnirun** script was invoked at boot time from the '**/etc/rc.local**' script. This option causes some one-time clean-up activities to be performed, such as log file maintenance.

**-core** This option, when specified, will allow all of the **pnid** daemons that are started to create '**core**' files if they should abnormally exit. This is used to debug the **pnid** program.

**-debug** This will cause a **pnid** daemon process to be run interactively with debug out sent to standard error. This has the effect of invoking the daemon with the **-t** and **-d** options.

**-down** This will cause the named instance of the **pnid** daemon process to exit if it is currently running. In this case, you should specify a daemon instance by name: **pni0**, **pni1**, etc:

```
/etc/pni/pnirun -down pni0
```

**configfile**
A list of interfaces can now be specified. For example, if '**pni0**' was specified, and a file named '**/etc/pni/config/pni0.config**' exists, then a **pnid** daemon will be started with that configuration file.

If the **-all** option was specified, then the **pnirun** script will build a list of configuration files. A **pnid** daemon will be started for each configuration file. The configuration files are located by the following process:

1. If the file '**/etc/pni/config/autostart**' exists, then it is expected to exactly contain a list of configuration file names, one per line. No other punctuation or commentary is allowed.
2. Else, if a directory '**/etc/pni/config**' exists, all the files of the form

   **'/etc/pni/config/*.config-auto'**

   are treated as **pnid** configuration files.

The usual case is the second; where all of the

**'/etc/pni/config/*.config'**

files are used and a **pnid** daemon started for each.

Note that the **pnirun** script is used for *outbound* instances of the daemon, where the local host is initiating some connection to a remote network access server. Inbound connections are handled by the '**/etc/pni/pnilogin**' script.

# 14 Troubleshooting and trivia

## 14.1 PNI not working - now what?

The *very* first thing which you should do when encountering problems with the PNI software to look at the log files. Since the **pnid** daemon process runs detached from any particular user on the system, it can only leave messages in log files to advise the system administrator and users of the host of any problems it might be having.

Most of the error messages, warnings and other commentary will appear in '**/usr/adm/pni.log**'. More serious errors and messages will also appear in '**/usr/adm/messages**', as well as on the system console '**/dev/console**'. You can use any text editor to view the two files in the '**/usr/adm**' directory, and the Console Tool from the workspace to view messages displayed on the system console.

The location of the log files are specified in the '**/etc/syslog.conf**' file. This file is modified when the PNI software was installed to add these lines:

```
# PNI::START::1.0
local5.debug          /usr/adm/pni.log
local5.warning        /dev/console
# PNI::END::1.0
```

The **local5** syslog facility is used for all logging messages generated by the **pnid** program. It is not possible to change this logging facility to another value.

## 14.2 What about HostManager?

The **HostManager** application is used to configure the "standard" network interface on NEXTSTEP system. In most cases, you do *not* want to use it to assign IP addresses, names or default routes for any network interfaces created and managed by PNI.

## 14.3 Problems with netinfo, lookupd and Mach messaging

There are a class of problems which you might experience which are caused by software not dealing well with network interfaces which were created *after* the program was started. Most of the software running on UNIX platforms were written with the assumption that network connectivity is a static state of affairs (other than when there are actual network failure, in which case some are not so graceful anyway).

For example, the **nmserver** daemon is responsible for forwarding Mach IPC messages between computers on a network. When the nmserver daemon is started at system boot time, it examines the configuration of the active network interfaces and initialized appropriately. Should additional network interfaces be added, it is necessary to "poke" the daemon and have it re-initialize itself. An example of this can be seen at the end of the '**/etc/rc.net**' shell script, where a USR2 signal is sent to the process to cause it to be reinitialized.

## 14.4 Kernel panics

If you are using PNI on a Intel based NeXTSTEP system, and are experiencing system crashes due to a kernel **panic**, you are probably experiencing a bug in the NEXTSTEP serial driver. This has been reported on NEXTSTEP 3.2 systems, especially when using PNI in Network Access Server mode.

The solution to this problem is to obtain the MUX driver from one of the NeXT FTP archive sites, and use it instead of the default serial driver.

## 14.5 Common error messages

If you see a message of the form **PNI pni: runt packet 4 bytes dropped**, this is the PNI encapsulator telling you that it has been handed a packet which is smaller than the minimum size of an IP datagram header. When this is the case, the PNI encapsulator doesn't even attempt to hand the erroneous packet the operating system kernel, but drops it and logs a message.

Why would this occur? One reason, which would be consistent with seeing only a *very* few of these messages could indicate a noisy or error prone serial link where an "extra" SLIP framing character may be been detected. The more likely case, however, is probably due to the reception of a TCP segment which was sent using Van Jacobson TCP header compression (CSLIP), but the local PNI software was not configured to use CSLIP, but rather, plain SLIP. Thus, the small header compressed packet was handed to the PNI encapsulator without having the compressed IP and TCP header replaced with a complete header.

## 14.6 Serial Device Troubles

## 14.7 Dialing Script Troubles

To aid debugging of dialing scripts, you can invoke the **pnid** daemon directly rather than using the **pnirun** shell script. The additional debugging becomes available by using the **-t** option to put the daemon into *test* mode. In this mode, messages which are usually logged via syslog will be emitted on standard output and standard error. Also, higher debug levels for the TTY encapsulator will cause the Dialer scripts to log data received and transmitted to standard output.

A typical invocation would be:

```
/etc/pni/pnid -C -c -n pni0 -t -d -f /etc/pni/config/pni0.config
```

or, slightly more simply, via the **pnirun** script:

```
/etc/pni/pnirun -debug pni0
```

Higher debug levels for the encapsulators can be set by using the **debug** subcommand in the config file, after the list of **stack** commands:

```
stack PNI       pni
stack FILTER    filter
```

```
stack SLIP      slip
stack TTY       tty

tty debug 8

proc LINK_start { encap } {
    log "LINK $encap connected"
}

proc LINK_stop { encap } {
    log "LINK $encap disconnected"
}
```

The values passed to the **debug** subcommand correspond to the *syslog* debug levels:

**7** Debug

**6** Informational

**5** Notice

**4** Warning

**3** Error

**2** Critical

**1** Alert

**0** Emergency

It is possible to have values larger than 7; these enable even more extensive debug output in some cases. Debug values at level 7 or greater are *not* recommended for normal production use of the software due to the volume of debug and trace messages that may be generated.

Each encapsulator has its debug level set independently of the others.

Log messages will be written to '**/usr/adm/pni.log**'. This can be changed by editing the '**/etc/syslog.conf**' file.

## 14.8 Modem Troubles

Make sure that you are using the correct cable to your modem. Specifically, on NeXT hardware, a Macintosh serial cable *will not* work on 68040 systems.

Due to hardware constraints, you can't do hardware flow control on 68030 systems, where you likely need it the most.

In most cases, you'll want to run the serial port speed at the highest speed possible to take advantage of any compression that's done in the modem (via MNP5 or V.42bis). This means you *must* use the '**/dev/cuf***' devices and the modem *must* be configured for hardware flow control. The dialing scripts provided should do this, if you need to write your own, you'll need to make sure this is the case, too.

Make sure that the modem and cable is configured so that the DCD line (Data Carrier Detect) from the modem is available at the serial port on the correct pin. This is how the **pnid** process checks to see if the modem is still connected or if the line has been disconnected.

### 14.8.1 TTYDSP related problems

If you are using TTYDSP, you should ensure that you have maintained the modified version of the **loginwindow** program which is created when the TTYDSP package is installed. This modification is necessary since the **loginwindow** program resets the sound driver when each user logs in. Since the TTYDSP package uses the sound driver to communicate to the DSP and the DSP's serial port, this will disrupt the use of the serial port.

The **loginwindow** program may be replaced when you upgrade from one operating system release to the next.

## 14.9 Network Problems

Can you ping yourself? Can you ping the other end of the link? Can you ping systems past the other end of the connection? If you can only get so far, then you have routing problems. Most of the time, your packets can get to where they're going with no problem; it is the returning traffic that can't find its way.

If you are a remote, standalone host dialing in over a SLIP connection, you probably *do not* want to be configured as being on a NetInfo network.

If you try *very, very* hard, you can probably configure things so that **-NXHost** works. But you'll find that the performance over a SLIP link is *so* bad, its unusable. I have yet to find *anyone* that finds otherwise.

### 14.9.1 Routing Issues

To effectively use the PNI package to communication with other hosts, it is necessary to arrange for proper routing to be in effect. There are two aspects to this: the local routing configuration must be set so that traffic can be correctly transmitted, and the routing configuration on the remote systems must be set so that traffic can find its way *back* to the local system.

### 14.9.2 Domain Name System Issues

A common desire when using the PNI package is to be able to communicate with many different hosts on the Internet. This usually requires the use of the Domain Name System to resolve domain names into IP addresses.

By default, the NEXTSTEP systems will only query the NetInfo directory service, which contains only local host information. To enable the use of the DNS on NEXTSTEP systems, you must create the '**/etc/resolv.conf**' file. Once this file has been created, any attempts to lookup host names will result in a DNS query if the name does not exist in NetInfo. See "man resolver" for more details.

Of course, if you can't contact external name servers because your SLIP link is down *most* of the time, having a '**/etc/resolv.conf**' file may be troublesome at those times. Later versions of PNI will delete any default route which it installed when link becomes inactive which *should* cause the resolver to give up quickly with a "Host Unreachable" error message. Note that in this case, you will trigger a bug in NeXT's **lookupd** daemon - it incorrectly caches DNS "Temporary Failure" returns (*e.g.*, can't contact name server) as "Host Unknown" errors for some period of time. This means that even *after* the link comes back up, you may still get cached "Host Unknown" errors for a short period of time, or until the **lookupd** process is restarted.

## 14.10 Questions and Answers

### 14.10.1 Weird telnet operation

Here's an bug report that was received regarding strange behavior of telnet sessions over a PNI SLIP link:

```
I've run into what appears to be a bug with PNI-1.11.  The problem is
that, under certain circumstances which I will describe below, <cr>
(and very occasionally <lf>) characters disappear.

Here's the circumstance where I noticed the problem.  My home NeXT
(running NS 3.2) is running PNI as a client, establishing a SLIP
connection to an Annex in my department.  I have a Sun Sparcstation 1
at my office running SunOS 4.1.3 and X-windows.  The problem occurred
with a telnet session from an xterm on the Sparc to the NeXT, via the
Annex and SLIP link.

Here are the symptoms:  If I do an "ls" on a medium-sized directory,
the first dozen or so lines are fine but after that many of the <cr>'s
are missing, so the lines
                          continue on the screen
                                                 like this.  I'm
running SLIP with an MTU of 256, which is the default on our Annex.
The dropped <cr>'s seem to always occur after the first 256 bytes.

After further experimentation with ls on very short directories, ls on
very long directories, and cat of a rather long file, I've come to the
conclusion that the errors always occur on the last packet of a burst
but never before.  The cat of the long file was correct, screenful after
screenful, until the very end, leading me to believe that the problem is
NOT with flow control on the modem.  (I am using hardware flow control.)
On the other hand, all 3 <cr>'s were missing in a 3-line "ls" listing of
a small directory.

How could this occur?  My conjecture is that PNI handles short bursts
differently than long ones in an attempt to improve responsiveness and
that that section of the code somehow loses the <cr>'s.  Of course, if I
enable "Auto Linefeed" mode in my xterm window, then the missing <cr>'s
are not noticed and everything appears to be working fine, which could
explain why it hasn't been reported before.  Also, I don't notice any
```

```
problems telnetting *out* of my NeXT to my Sparc, just inbound.  I don't
know whether or not this problem affects other network services such as
FTP.
```

This isn't a bug in PNI. PNI isn't aware of any of the actual characters being transmitted in the packets going by, and certainly doesn't muck with the contents of any TCP connections which is carries. PNI is definately not doing anything with CR or LF characters in the user's data stream. If it did, then the TCP checksums wouldn't work, and the connection would hang or reset.

This is indictive of an application level program which is being provoked by either timing differences (slower network connectin), or just an incompatibility.

There are two potential problems:

What's going on here could be a timing interaction with the TTY driver. You are probably using a shell like bash or tcsh on the remote system. This shell normally operates in cbreak or raw mode, but switches to cooked mode to run programs. When the tty is switched from one mode to another, pending characters in the queue are flushed. In this case, before they are transmitted to the network.

The other more likely case is that they telnet daemon isn't implementing telnet line mode correctly, which includes setting up the tty into the correct mode when switching from raw mode to cooked mode, etc.

One way to test this is to use rlogin and see if it operates differently; it doesn't try to be quite a clever at the telnet daemon is.

### 14.10.2 `pnistat` questions

Consider the output for an invocation of the '**`/etc/pni/bin/pnistat`**' program invocation: **`pnistat`**

```
                          Encapsulated      Decapsulated
   Type   Encap state      Bufs     Bytes    Bufs     Bytes
   PNI*     pni phase2    27192   2018308   38633   8455145
FILTER*  filter phase2    27194   1148244   38635   8455657
  SLIP*    slip phase2    27195   1148284  146807   7207990
   TTY*     tty phase2    27196    290858  146813   7208499
```

There are several bits of trivia to be gleaned from this example.

First, note that ascending number of buffers listed under '**`Encapsulated`**'; this is an artifact of the measuring process. At the time, a bulk file transfer was taking place, with constant input. The statistics are gathered one encapsulator at a time, and during the reporting process, packet traffic was in progress. The packets being transmitted (encapsulated) at the time consisted of TCP ACK segments.

Also notice that the encapsulated byte count of the '**`PNI`**' encapsulator ('**`2018308`**') is somewhat higher than that of the '**`FILTER`**' encapsulator ('**`1148244`**'). This is reflects the fact the that PNI kernel driver returns a fixed length header of 32 bytes on every packet transmitted by the kernel. This header is stripped off before handing the IP packet up to the next encapsulator.

Note that the Van Jacobson TCP header compression algorithm is resulting in a significant savings of bytes transmitted over the serial line; consider the difference between the '**SLIP**' encapsulator byte count, '**1148284**' and the '**TTY**' encapsulator byte count, '**290858**'. This reflects the reduction of a 40 byte TCP ACK segment down to just a few characters per packet. A similar reduction can be seen in the '**Decapsulated**' statistics, which are large in this example because bulk movement of data into the host. The percent reduction is not as large since the 40 byte overhead of the TCP packet is relatively small compared to the actual user-data being carried.

## 14.11 Internal Organization

The general architecture of the PNI package has already been described in another part of this document (See Appendix A [Architecture], page 80). Here, we'll present some of the more more nitty-gritty details of how the software fits together and functions.

## 14.12 Encapsulator Internals

Encapsulators are implemented as Objective-C classes which are dynamically loadable from bundles in the '**/etc/pni**' directory. The bundles all have an extension of '**.encap**' and are, by convention, all in upper case. Within each bundle directory, there can usually be found three files.

There is an object file which contains the compiled (multiple-architecture) binary code for the class which implements the Objective-C class. All encapsulators are subclasses of a common encapsulator class. Currently, the interface specifications of the super-class are unpublished and subject to change.

There may also be two TCL files in the bundle directory. One TCL file contains code which is loaded *once* the first time a particular encapsulator is instantiated by the **pnid** daemon process. The other is loaded *each* time the encapsulator class is used. In usual configuration, it is not likely that the encapsulator will be used more than once per configuration so each file of TCL code is loaded once.

For example, for the '**SLIP**' encapsulator the files are:

'**/etc/pni/SLIP.encap/SLIP**'
: The binary file which contains the compiled code and data.

'**/etc/pni/SLIP.encap/SLIP_Init.tcl**'
: The per-class encapsulator initialization file. When this file is loaded, the global TCl variable **encapType** is set to the encapsulator type; in this example **SLIP**.

'**/etc/pni/SLIP.encap/SLIP_objInit.tcl**'
: The per-instance encapsulator initialization file. When this file is loaded, the global TCL variables **encapType** and **encapName** are set to the encapsulator type (in this example **SLIP**) and the name of the encapsulator as specified on the **stack** command.

The usual contents of these initialization files are procedure definitions. In particular, the following TCL procedures are expected to be defined and will be invoked if defined when appropriate (in this example, for the '**SLIP**' type encapsulator named '**slip**' in the configuration)

**slip_start{ }**

**slip_stop{ }**
**slip_configure{ encapName }**

## 14.13 pnid internals

*This section still under construction. . .*

For a starting point, look at the '/etc/pni/pnid.tcl' and '/etc/pni/support/init.tcl' files.

# Appendix A Architecture

This section will describe the general shape and architecture of the PNI package. While it is not necessary to know all of the details described in order to use the software, it will give you a better sense of how it PNI is put together and how it might be used.

## A.1 Network Interface Architecture

A *Network Interface* is the logical connection point of your computer system to a network. It corresponds to the logical "hardware" device which allows the flow of data into and out of your computer system. The most common instance of a network interface on a computer system or workstation is the *Ethernet* interface.

The term *Network Interface* is also used to refer to the software abstraction or data structure used by the operating system running on your UNIX computer system or workstation. The network interface has some state associated with it (such as an IP address, subnet mask, broadcast address) and is referenced by other operating system data structures, such as the routing table which is consulted to determine the path to transmit network traffic for a particular destination.

The TransSys PNI product provides a way of creating additional network interface abstractions or data structures to augment the usual Ethernet network interface. It also has software which handles the network traffic being sent on these network interfaces and that traffic which will be received on those network interfaces. Network interfaces are named; for instance, the Ethernet interface is usually named **`en0`**; network interfaces created by the PNI software are named **`pni0`**, **`pni1`**, etc.

The TransSys PNI product consists of a number of daemon processes, each of which corresponds to a particular logical network interface. Not all network interfaces need be active at once; when traffic begins on a particular network interface a connection can be established on demand ("dial on demand"). *Please note that in the current beta version of the software, dial-on-demand is not yet available for use.*

## A.2 PNI Software Architecture

When configuring and operating the PNI software, it is often helpful to understand the underlying architecture of the software. Often, there is some critical piece of information missing when trying to communicate a complicated set of information. In many cases, this critical information is the underlying model which the software was written to conform to. Or *"What was he thinking when he did it like that?"*

There are two main software components which constitute the PNI package:

- **`pni_reloc`** – the loadable kernel device driver.
- **`pnid`** – the user-level support daemon process.

## A.3 `pni_reloc` kernel driver

The **`pni_reloc`** kernel driver is automatically loaded into the UNIX operating system. It's task is to create a standard network interface abstraction for the existing kernel TCP/IP networking code to use. It very similar to the existing '**`en0`**' Ethernet interface in the sense that network code in the operating system uses the '**`pni`**' interfaces the same way that it might use the ethernet ('**`en0`**') or system loopback ('**`lo0`**') network interfaces.

The main difference between the standard network interfaces and the '**`pni`**' network interfaces is that there is no actual hardware resource directly associated with the '**`pni`**' network interface. Rather than having some hardware for the driver to use, we use a *virtual* hardware interface.

This is actually quite similar to the UNIX **`pty`** (pseudo teletype) devices that UNIX uses. There are not actual serial devices associated with the **`pty`** devices, but another, special interface which is used to control a simulated serial device. This is how remote network logins work; your session is associated with a pseudo tty device; it is controlled by a telnet daemon process which interfaces to the operating system to read and write characters to and from the network. There network character streams are then made available via the **`pty`** device, and the UNIX shell and other commands believe that they are attached to a terminal type device.

The **`pni_reloc`** device uses a similar abstraction (in fact, *pni* originally was an acronym for Pseudo-Network Interface). For each pseudo network interface device that's created (e.g., '**`pni0`**') there is a corresponding character special device ('**`/dev/pni0`**') which is used to control the network interface and transfer the data. When the operating system transmits a packet of data to the network via the '**`pni0`**' network interface, that transmitted package is available to be *read* on the '**`/dev/pni0`**' character special device file. Conversely, when the '**`/dev/pni0`**' file is written to with the correctly formatted data, that data appears as *input* from the '**`pni0`**' network interface.

So far, it sounds as if the pseudo network interface kernel driver isn't really doing all that much for us; it's mostly just shoving the data that come in from one end (output from kernel networking code) and sending out the other (to be read from the '**`/dev/pniX`**' device file). Actually, that simplicity is one of the primary design goals for the kernel driver; it is as simple as it can possibly be. This simplicity is a great advantage in a kernel device driver - the less the kernel driver code needs to perform, the less time it will take to debug that code. Malfunctioning kernel drivers, rather than just dumping a core file like an application might do, will *panic* or crash the entire system.

While just implementing SLIP or CSLIP in a kernel driver is not very difficult (and TransSys has already done this – our **TransSys Dial-Up IP** package for NeXT 68040-based systems), more complex protocols such as PPP and other features (such as packet filtering) become somewhat unwieldy when implemented within the framework of a kernel device driver. The intent of the **`pni_reloc`** driver was to create as minimal a kernel driver as possible (or necessary), and implement the remaining functionality in a user-level (that is, not a kernel-based) process.

The other advantage of this approach is that the user-level process can be written in a portable fashion, insulated from operating system changes because it uses a defined (and presumably unchanging) API to get its work done. Only the (much smaller) kernel driver actually needs to be "ported" as the operating system has major upgrade or when moving from one type of platform (NEXTSTEP/UNIX/Mach) to another (SunOS? AIX? Who knows?).

## A.4 `pnid` daemon process

The **`pnid`** program is what is on the “other end” of a pseudo network interface. It opens the ‘**`/dev/pni0`**’ (or ‘**`/dev/pni1`**’, etc) character special device file, and reads data from the file (output packets from the corresponding ‘**`pni0`**’ network interface) and writes data to the file (input packets to the ‘**`pni0`**’ network interface).

The **`pnid`** program is the bridge between the pseudo network interface and some real hardware or transport for the packets. For example, the **`pnid`** program might also have open a tty serial device like ‘**`/dev/cufa`**’ so that it might receive and transmit network packets over a serial port to a modem.

If you have arranged to use more than one PNI pseudo network interface (perhaps because you have multiple PNI-based network links), you will have a separate and distinct **`pnid`** daemon process for each active pseudo network interface. Strictly speaking, the architecture of the software doesn’t *require* a daemon process per pseudo network interface; but implementation and operational issues make the one process per interface approach an attractive approach to use.

The **`pnid`** process is implemented in a rather novel way. Since the kernel-level pseudo network interface driver is so simple in function, all of the *interesting* functions need to be implemented in the **`pnid`** daemon program. The daemon is built from a number of well defined, reusable modules which can be arranged and configured in different ways depending upon the circumstances.

The best way to describe these modules is to explain how they are used to build a network connection and how the modules handle and modify the data corresponding to the packets flowing over the network connection. Let’s enumerate the steps which occur for a SLIP connection’s flow of data.

1. First, obviously, is the fact that a network interface must exist which is the source and sink of the packets to be sent and received. This is function is implemented by the **`pni_reloc`** kernel driver.
2. There must be a mechanism for obtaining the packets which are being transmitted on a network interface. This is done by opening and reading the ‘**`/dev/pni`***X*’ character special device file. This is done by a module in the **`pnid`** process that’s called **`PNI`**.
3. The SLIP protocol specifies certain conventions for encoding and framing an IP datagram so that it may be transmitted over a serial link. This encoding and framing process consists of adding a frame delimiter character to separate one packet from the next, and a mechanism for escaping the special characters that are reserved for framing purposes. This function is implemented by the **`SLIP`** module. The SLIP module processes as *input* packets which are to be sent and produces as *output* a stream of characters which are the encoded and framed input packets.
4. Finally, the character stream of encoded and framed packets is actually transmitted on a serial port. On UNIX systems, this is done by opening a tty-like device (for example, ‘**`/dev/cufb`**’), setting the tty device in RAW mode, and then writing streams of characters to that device. This is done by the **`TTY`** module.

Of course, for incoming packets the same functions and operations are required, but simply applied in reverse order in an inverse fashion.

Each of these functions are performed in essentially a linear fashion; the “data” starts at one end, and is successively processed or transformed as it moves along. Each module is referred to as an *Encapsulator*; encapsulators have standard interfaces which allow one encapsulator to use any compatible encapsulator as its “input” or “output”.

The design of the **pnid** program is such that each instance of the daemon services a particular network interface. So if multiple pseudo network interfaces are configured for dialing-out, there will likely be more than one **pnid** process running.

Each encapsulator has well defined interfaces for input and output, and thus a set of them can be arranged in order to provide the desired function. The encapsulators are actually implemented as Objective-C loadable classes. Future encapsulators can then easily be accommodated so long as they implement the standard interfaces.

# Appendix B Bibliography

Some references to useful information are included here.

**TCP/IP** *TCP/IP Network Administration by Craig Hunt*, O'Reilly & Associates, Inc. 1992. ISBN 0-937175-82-X

**DNS** *DNS and BIND by Paul Albitz & Cricket Liu*, O'Reilly & Associates, Inc. 1992. ISBN 1-56592-010-4

**Tcl and Tk**

*Tcl and the Tk Toolkit* by Dr John K Ousterhout. Addison-Wesley Publishing 1994. ISBN 0-201-63337-X. A clear and readable introduction to TCL (which is used extensively in PNI) and Tk.

**Practical Programming in Tcl and Tk**

*Practical Programming in Tcl and Tk* by Brent Welch. Draft of book to be published by Prentice hall. A tutorial-style introduction to Tck and TK. Draft version available via anonymous FTP from `parcftp.xerox.com` in '`/pub/sprite/welch/tclbook2.1.ps.Z`'.

# Concept Index

# Function Index

# Variable Index

## C



## E

# Table of Contents