Testing Protocol Implementations with *pkt*

Rusty Eddy
Ramesh Govindan
USC/Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina Del Rey, CA 90292
Phone: `+1-310-822-1511`
Fax: `+1-310-823-6714`
Email: `eddy,govindan@isi.edu`
WWW: `http://www.isi.edu/ eddy/pkt`

Abstract

pkt is a tool that allows users to symbolically manipulate the contents of network protocol packet headers, and control the generation and reception of packets. Such a tool has, in our experience, proven to be invaluable for debugging and testing protocol implementations. In particular, we used *pkt* extensively while implementing PIM-SM [3].

pkt combines the scripting features of Tcl [15] with the low-level packet capture facilities provided by the *pcap* library. This paper describes the design of *pkt* and chronicles our experiences with this software.

1 Introduction

In the Internet today, routing protocols propagate network reachability or link state information and thereby implicitly govern the flow of unicast traffic. Several routing protocols are currently in use (*e.g.,* RIP [8], OSPF [13], BGP [18]). In the near future, we expect the introduction of native, network-layer multicast in the Internet. Several multicast tree construction protocols are currently being designed and prototyped (*e.g.,* PIM-SM [3], CBT [1], BGMP [4] *etc.*). These protocols use the reachability information provided by the unicast routing protocols to construct distribution trees from one or more sources to a group of receivers.

Even though the principles behind them may be relatively simple, today's unicast and multicast protocol implementations can be quite complicated. There are several reasons for this. First, routing protocols are instances of complex distributed computations which strive to produce consistent loop-free paths in a dynamic Internet. Second, to allow fine grain administrative controls, routing protocol implementations contain several dozen configuration parameters. Third, routing protocol pairs may need to interoperate with each other at service provider boundaries. Examples of such protocol pairs include OSPF and BGP, or PIM and DVMRP.

Given the dependencies and interactions between protocols, and the user-specifiable configuration knobs in the protocol implementations, testing protocol implementations can be quite tedious. To test a particular feature of a routing protocol, implementors may construct a test network designed to explicitly test that feature. Then, they will configure the network appropriately, execute a number of test steps, and observe the resultant routing state. When a routing protocol implementation has been modified, the tester must repeat the above sequence of steps for every protocol function that may have been impacted by the modification.

Pkt is a tool that promotes a slightly different approach to testing. A user of *pkt* specifies, using an extension of Tcl [15], low-level packet generation events. The user has complete control[1] over the contents of the entire packet, including the network-layer (*e.g.,* IP) headers. The user may also express, using Tcl extensions, predicates defined on the contents of received packets. These simple primitives allow *pkt* to be used both as a diagnostic tool, and as a test scripting tool. Furthermore, as we show later in the paper, *pkt* can, in some cases, reduce the tedium of test network configuration.

Although *pkt* was designed as a diagnostic tool for our implementation of the PIM Sparse Mode multicast tree construction protocol, it is applicable to several other low-level network control protocols. This paper describes the design and implementation of *pkt*. Section 2 illustrates the motivation for the design of *pkt* using an example test case, and Section 3 describes the software design and usage of *pkt*, Section 4 describes how we've used *pkt* and other potential uses. Section 5 Mentions other network test tools comparable to *pkt*. Section 6 Provides a conclusion and future direction.

2 Motivation and Background

In this section, we show how an implementor might plausibly design a test network for verifying whether a routing protocol implementation converges after link failure. We then illustrate how *pkt* can be used to realize this same test, but with significantly fewer resources. This demonstration motivates various aspects of *pkt* design and brings out the flexibility of *pkt* for debugging, diagnosis and testing.

[1] With one caveat. The user may not have complete access to the link layer header information.

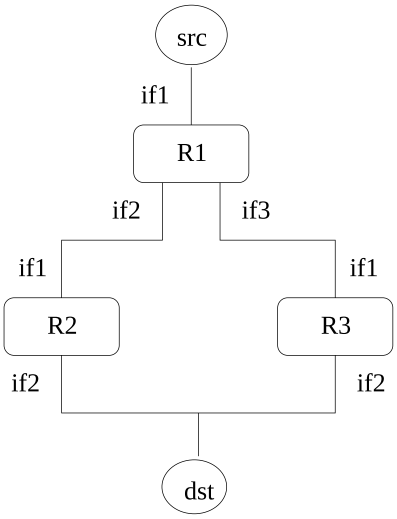

Figure 1: Example Test Network

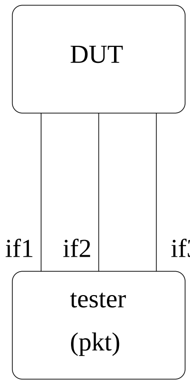

Figure 2: Emulator

2.1 Example Test Network

Hop-by-hop dynamic routing protocols react to link failures by finding alternate, but possibly higher cost, paths through the network. How might an implementor verify whether a particular routing protocol implementation correctly converges to an alternate path? *One* way to do this would be to construct the test network shown in Figure 1. This test network contains two parallel paths between the data source and destination, *src* and *dst*, respectively. *r1*, *r2* and *r3* are intermediate routers which run instances of the routing protocol implementation.

The routing protocol establishes the preferred path between *src* and *dst*, and selects an alternative path if the preferred path becomes unavailable (due to a link failure, for example). Distance-vector protocols, such as RIP, do this is by having *r2* and *r3* each advertise their *cost* of forwarding traffic to *dst*. *r1* adds the costs advertised by *r2* and *r3* to it's own cost of accessing each respective router. This will determine the lowest cost path toward the destination. For example, if *r2* and *r3* each advertise to *r1* a cost of 1 to *dst*, and if the cost for *r1* to access *r2* is 10, then *r1* can send data to *dst* via *r2* with a total cost of 11. Alternatively, if the cost of *r1* to send to *r3* is 20, then *r1* can send data to *dst* via *r3* with a cost of 21. As long as *r2* is reachable and *r2* can access *dst*, *r1* will choose *r2* as the preferred path for sending data to *dst*. In the event of a link failure on *r2*, after a possible *convergence* period, *r1* will start forwarding data via *r3*. To test even this simplistic scenario, the implementor must:

1. Set up the test network shown in Figure 1.
2. Start all routers and wait for the route tables to converge.
3. Send a constant stream of data from *src* to *dst* and verify the data flow.
4. Shut down one of the network interfaces on *r2*, disrupting data delivery.
5. Wait to see the data once again delivered to *dst* via *r3*.

This approach to testing protocol implementations is resource-intensive and tedious. Even testing such a simple feature requires three routers, two hosts and enough network equipment to support the configuration shown. If the test fails, the implementor may need to repeat the test several times, with possibly finer grain control over device actions and event timing. Some of the tedium may be alleviated by implementing ad-hoc scripts to automate some aspects of the test.

2.2 The Convergence Test Using *pkt*

As we have described earlier, *pkt* provides Tcl extensions to:

- Arbitrarily manipulate the contents of a packet and control the relative ordering of packet-generation.
- Test received packets for specific contents.

In this section, we illustrate how these two simple primitives enable us to test the convergence example using *pkt*. In a later section, we describe these primitives and how they are implemented in greater detail.

pkt primitives provide low-level access to packet contents, and fine-grain control over packet generation. With these features, *a single instance of* pkt *can be used to send data or protocol control traffic from arbitrary network addresses*. This allows us to **emulate** a large network using one instance of *pkt*, thereby reducing the resources and tedium of test network generation.

Figure 2 shows how the routing convergence test can now be implemented using two devices: a *tester* which runs *pkt*, connected to the *device under test (DUT)*. In this example, the DUT is equivalent to *r1* in Figure 1. The tester is directly connected to the DUT by three parallel links; using these three links, the tester emulates traffic exchanged across the three interfaces of *r1*.

1. `if1`: generate a routing protocol message that advertises reachability to *net3* from *r2*
2. `if2`: generate a routing protocol message that advertises reachability to *net3* from *r3* at a higher cost
3. `if3`: generate a stream of data packets destined for *net3*
4. `if1`: wait to receive at least one data packet on this interface
5. `if1`: generate a routing protocol message that advertises *non-reachability* to *net3* from *r2*
6. `if2`: wait to receive at least one data packet on this interface
7. terminate

Figure 3: Convergence Test Pseudo Code

What is the structure of a *pkt* program that runs on the tester? We describe this program in the pseudo-code of figure 3.

2.3 Motivation for *pkt*

We have used the above example to illustrate how a tool like *pkt* can be used to elegantly test separate pieces of protocol functionality. As we have seen, *pkt* can, in some cases, obviate the need for complex test network generation. The functionality that *pkt* provides is useful in other ways:

- It gives greater control over the timing of packet generation events. In a test network, on the other hand, the timing of events may be determined, for example, by when the various devices are started up.
- It can generate erroneous packets or packet sequences. This is useful to test the robustness of the implementation to a variety of inputs.
- Finally, *pkt* can be used to verify relative temporal ordering of received packets. In the example above, this feature allows the tester to verify the causality between advertising non-reachability of *net3* from *r2* and receiving data on `if2`.

The basic functionality of *pkt* (user controlled low-level packet generation) can be provided in several different ways. We imposed three additional requirements on the design of *pkt*. First, *pkt* must allow users to symbolically manipulate packet contents. Thus, users must be able to say the equivalent of "set the ttl field of the IP header to 127", rather than "set the 9th byte of the packet to 127". Second, *pkt* must be easily extensible to new protocols and protocol headers. In particular, users must be able to symbolically specify the layout of packet headers. Finally, *pkt* users must be able to rapidly prototype and modify tests. This functionality is particularly useful when the developer is debugging protocol implementations.

These considerations led us to design *pkt* as a Tcl extension, similar to expect [7]. Not only does the choice of Tcl allow us to meet the requirements described above, but it also leverages the large body of other Tcl extensions that can be seamlessly integrated into a *pkt* program. For example, we have found it very useful to incorporate SNMP primitives provided by the Tcl extension *scotty* [20]. Furthermore, tests encoded as Tcl scripts are reusable and can be grouped into test suites, such as regression tests (Section 4).

3 *pkt* Design and Implementation

pkt can be described as a simple packet scripting language whose primitive constructs are *protocol headers*, *packets* and *devices*. Users can define new abstract data types (i.e. different kinds of packet headers) and thereby extend the functionality of *pkt*. This section describes these constructs. In the next section, we discuss some implementation details.

Pkt, like most other Tcl extensions, provides a shell that will run interactively for debugging or as a stand-alone program when executed as a script. *Pkt* will also be available as loadable Tcl extension [2].

3.1 Protocol Headers

A header is a construct representing the fields of a protocol header. Several predefined header types exist in the *pkt* programming language: these include the IP header, the UDP header, and the IGMP [6] header. To create a particular instance of a header, a user might employ the following syntax: `header new` command.

```
header::new ip iphdr
header::new udp udphdr
```

These directives create an IP header labeled `iphdr` and a UDP [17] header labeled `udphdr`. *pkt* assigns default values for header fields.

pkt also provides primitives to store and retrieve header field values. For example, the following examples show how a user might assign various values to the field of header instances:

```
header::fields iphdr -ttl 1 \
        -dst 225.1.1.1 -src 10.1.1.1

header::fields udphdr -sport 1234 \
        -dport 5678
```

A `header::field` directive may be used to retrieve specific field values. This command is useful, for example, to test the contents of received packets.

In our current implementation, headers are represented as Tcl associative arrays, with indexes corresponding to the header fields. Using this Tcl-native data representation for headers enables us to 1) avoid C language implementations

[2] this is currently not true, but will be, Real Soon Now

of header manipulation functions, and 2) add new header types by defining using the `header define` procedure. The following example shows how the UDP header type is defined in *pkt*:

```
header::define UDP ''sport dport len sum''
header::field UDP::port -bits 16
header::field UDP::len  -bits 16 -default 4
...
```

The `header::define` subcommand implicitly defines the symbolic names associated with the fields of the UDP header and thier relative ordering. The `header::field` command can be used to set the bit width of the field, default value, and a printer for producing field specific data formating. This subcommand informs *pkt* the complete layout of the header; with this information, *pkt* transparently generates the "wire" representation of the packet, extract the header from such a representation, and perform byte-order conversions.

3.2 Packets

A packet is a construct representing a sequence of protocol headers. A user creates an instance of a packet header using the `packet::encode` command. For example, the user might create a data packet labeled `p1` using the headers defined in the previous section as follows:

```
packet::encode p1 iphdr udphdr 01234567
```

Conversely, having read a packet `p2` from a device (Section 3.3), the user can extract a header from `p2` using the command:

```
packet::decode p2 IP iphdr2
```

This command extracts the IP header labeled `iphdr2` from `p2`.

Several operations are defined on packet instances, a complete listing is available in with the software, and the website. Each operation on a packet is represented as a subcommand of the `packet` command. For example, the command

```
packet::copy p1 p2
```

creates a new packet instance labeled `p2` whose contents are identical to `p1`. Similarly,

```
packet::delete p1
```

deletes the instance labeled `p1`, and

```
packet::len p1
```

returns the total length of `p1`.

Internal to *pkt*, a packet instance is represented in binary form. Specifically, the `packet::encode` command constructs a *packet string* (simply a Tcl string containing the hexadecimal representation of the entire packet) from the associative arrays representing packet headers. It then invokes a C function that converts this intermediate packet string to the binary representation of the packet. This choice of a Tcl-native representation of a packet allows users to avoid writing C functions for converting individual headers to their binary forms.

3.3 Devices

A *device* type represents a network interface attached to the node *pkt* is executing on. A device instance is opened for read/write using the `device open` command, as in:

```
device::open if1
```

which opens the device labeled `if1` for future use. Several other operations are defined on devices. The following command writes packet `p1` to the device `if1`:

```
device::send if1 p1
```

and the following command assigns a label `p2` to the next packet[3] received on device `p1`:

```
device::read if1 p2
```

also, a user may attach a *filter* to a device:

```
device::filter if1 ''multicast and udp''
```

This command ensures that subsequent reads from the device only return UDP packets destined to a multicast address. A complete listing and all `device` subcommands is also provided in Figure ?? at the end of this document.

The syntax of the filter expression is identical to that of *tcpdump* [22]. In fact, *pkt* uses the *pcap* [19] library for packet capture. This library provides a generic functional interface to a variety packet capture facilities provided by different operating systems. Filters are provided by *pcap* to select packets that meet specific criteria, they can be applied to devices or packet instances. Filters are defined by the user as ASCII strings and compiled by pcap into packet matching programs (BPF [19] programs) [4].

3.4 Other Design Issues

This section discusses two other issues that arise in writing *pkt* scripts: controlling event execution times, and handling variable length protocol headers.

Several protocol actions, including packet generation, are triggered by timer expiration. For example, the PIM protocol join messages are periodically triggered by the expiration of its join timer. To model this, a *pkt* programmer needs to explicitly control the scheduling of protocol actions. Rather than define its own event scheduling mechanism, *pkt* uses Tcl's `after` command. *pkt* provides a second timing related feature, users can specify an optional timeout on the `device read` command; the absence of received packets during this interval may indicate, for example, an implementation error.

Defining and manipulating fixed sized protocol headers or fixed sized fields is relatively straightforward. How does *pkt* deal with variable sized packet headers? There are several design choices. Rather than explicitly providing support for this, *pkt* relies on the programmer to handle variable length

[3] This directive allows timeout, as an optional argument. If no packets are received within this time, the command returns the appropriate indication.

[4] Please refer to the `tcpdump` manual for a more complete description of the filter syntax.

headers. Specifically, the user is expected to define a variable length portion of a header as a separate header. This choice keeps the *pkt* header manipulation syntax relatively simple, at the expense of a possibly increased programming complexity. As for variable length fields, the `header::field` command provides the '-bits' argument, to set width of a given field. When the width of a bit field changes, the corresponding value of the field is adjusted and the length of the header is recalculated automatically.

3.5 Implementation Issues

Although *pkt* carefully tries to hide details of devices and filters, programmers must be aware of three operating systems issues when writing their packet programs.

First, the *pcap* library imposes the following restrictions on the *pkt* programmer:

- Once a device is open, packets matching its filter are buffered until they are explicitly read from the device. This can result in operating system buffer overruns on high traffic network segments. To deal with this, programmers may use the `device clear` command, reset the filter attached to the device, or simply open and close the device as needed.
- In some cases, the `device read` command may return outbound packets. This can happen, for instance, in operating systems that use BPF. To avoid erroneously processing such packets, the programmer must carefully set device filters to ignore outbound packets. Conversely, on systems with the `/dev/nit`[9] packet capture facility, the `device read` command will never return outbound packets, regardless of the filter attached to the device.

Second, *pkt* can provide the programmer access to link-layer header information. Specifically, a programmer may set, for some devices, the source link-layer address on outbound packets; *pkt* programs can thereby emulate several network interfaces from a single host. Conversely, *pkt* programs can examine, for some devices, the link-layer headers of received packets. Despite these features, *pkt* is not explicitly designed to allow testing of link-layer protocols.

Finally, when a *pkt* test computer emulates emulates numerous virtual devices, the programmer may have to explicitly configure the host to respond to ARP [16] [5] requests on behalf of the virtual devices. For example, in our convergence test example (Section 2), when the *DUT* attempts to forward the data packet to `if2` of the tester, it will fail to resolve the link-layer address. The *pkt* programmer must explicitly configure the tester to proxy ARP for destination address of the data packet. On UNIX systems, the following command suffices for this:

```
arp -s r2.test.net 02:03:04:05:06:07 pub
```

[5] For network types that require the Address Resolution Protocol, such as Ethernet. Other interface types, such as point-to-point links do not require this.

4 Experience with *pkt*

Even though Section 2 shows an example of a *pkt* program that is "equivalent" to a test network, we do not mean to suggest that *pkt* will replace test networks. Indeed, test networks may reveal unexpected protocol interactions; *pkt* is useful to test expected protocol behavior. Realistically, we believe that *pkt* is a useful addition to the suite of tools that protocol implementors use today.

pkt was developed to test our implementation of the PIM-SM [3] multicast routing protocol. Briefly, PIM-SM constructs shared multicast distribution trees routed at a Rendezvous Point, or RP. Multicast receivers send explicit *join* messages towards the RP; senders transmit data to the RP, which then redistributes data to the receivers along the distribution tree. Optionally, receivers may switch to a sender-rooted distribution tree for high data rate senders. We have implemented PIM-SM in GateD [2].

pkt proved invaluable in simplifying the debugging of our PIM-SM implementations. For example, to debug the protocol functionality that allows receivers to switch from a shared tree to a sender-rooted tree would have required constructing a test network. *pkt*'s network emulation capability helped us rapidly debug this feature with just two workstations. Furthermore, *pkt* enabled us to more carefully schedule packet generation events, so we could examine intermediate values of various protocol data structures. *pkt* was also invaluable in debugging another feature of PIM-SM: RP bootstrapping. In PIM-SM, candidate RPs announce their candidacy to a central bootstrap router; this latter router summarizes the collection of RPs and floods them to every router. *pkt*'s emulation capabilities greatly simplified the testing of this rather complex piece of protocol interaction.

We also used *pkt* for regression tests of router functionality. Based on experience with our implementation, and that of router vendors, the PIM-SM protocol evolved significantly. *pkt* was useful in verifying that the protocol changes did not introduce subtle bugs in existing protocol functionality. For example, the contents and interpretation of messages sent by the bootstrap router changed during the time we were implementing PIM-SM. *pkt* was useful to verify that our implementation of these modifications did not, in any way, break the construction of the shared multicast distribution tree.

Finally, we have used *pkt* for testing interoperability with other PIM-SM implementations (notably those from router vendors). Especially for this, *pkt*'s design as a Tcl extension has proved very useful. We were able to introduce *scotty* [20] SNMP primitives into *pkt* scripts; this helped us determine router intermediate states during our interoperability tests.

We can envision several other uses of *pkt*, although we haven't ourselves had occasion to use *pkt* in these ways:

Stress Testing *pkt* can be used to algorithmically generate random, high volume workloads for protocols. This is sometimes useful for triggering subtle timing errors in the protocol implementation.

Reproducing Error Reports *pkt* can be used to recreate errors reported from actual deployments. In particular, because *pkt* uses pap, it can be easily used to replay `tcpdump` traces.

5 Related Work

pkt is heavily influenced by other Tcl extension packages, such as *scotty* and *expect* [7]. Just as *scotty* is a Tcl extension for rapidly scripting network management tasks, and *expect* is a Tcl extension for scripting user interactions, *pkt* is a tool for rapidly scripting tests of protocol implementations.

The design of low-level packet capture facilities has received much attention [19]. *pkt* is itself not a packet capture facility. It provides higher-level access to such a facility; the power of *pkt* really derives from combining Tcl's capabilities for symbolic manipulation, and the *pcap* library's access to low-level packet capture.

pkt is most closely related to two other network emulators: Packet Shell [10] from Sun and the commercial product ANVL [14] from Midnight Networks. Packet Shell is also a Tcl-based and supports symbolic of packet contents. ANVL is a powerful well supported commercial product. Both products are integrated with low-level packet capture facilities. However, in our opinion *pkt* appears to provide easier extensibility. These products users are required to write C code to define new protocol header types. And in the case of ANVL, tests had to be written in C as well.

6 Conclusions

pkt is not a new idea. It combines two powerful existing ideas: Tcl's scripting and symbolic manipulation capabilities, and the flexible packet capture provided by several existing operating systems. Although we have focused on the use of *pkt* with routing protocol implementations, *pkt* is equally applicable for scripting datagram application-level protocols (e.g. DNS[12], NTP[11], DHCP[5]). The *pkt* software is freely available, and several protocol implementors groups, including the GateD Consortium have indicated interest in using our software.

pkt is a useful addition to the set of tools that are available for verifying the correctness of protocol designs and implementations. *pkt* can also be used for example, to allow users of a realistic network simulator, like *ns* [21], to incorporate real implementation within a simulation scenarios. That is, the device running the real implementation would form one node of the simulated network. *ns* could use the *pkt* primitives to generate and receive actual data and control traffic. Using this combination of emulation and simulation, implementors might study the behavior of their protocol in large-scale settings.

References

[1] *et. al.* A. Ballardie. Core based trees (cbt version 3) multicast routing – protocol specification –. draft-ietf-idmr-cbt-spec-v3-00.txt, March 1998.

[2] GateD Consortium. Gated routing software. http://www.gated.org.

[3] *et. al.* D. Estrin. Protocol independent multicast - sparse mode: Protocol specification. draft-ietf-idmr-pim-sm-specv2-00.txt, September 1997.

[4] *et. al.* D. Thaler. Border gateway multicast protocol (bgmp): Protocol specification. draft-ietf-idmr-gum-02.txt, March 1998.

[5] R. Droms. Dynamic host configuration protocol. Request For Comments 1531, Internet Directory Services, October 1993.

[6] W. Fenner. Internet group management protocol, version 2. Request For Comments 2236, Internet Directory Services, November 1997.

[7] D. Libes. Regression testing and conformance testing interactive programs. In *Proceedings of the 1992 Summer USENIX Conference*, pages 135–144, San Antonio, TX, June 1992. USENIX.

[8] G. Malkin. Rip version 2 - carrying additional information. Request For Comments 1723, Internet Directory Services, November 1994.

[9] Sun Microsystems. Network interface tap. Documentation for suns */dev/nit* packet capture facility.

[10] Sun Microsystems. The packet shell protocol testing tool. http://playground.sun.com/psh.

[11] D. L. Mills. Network time protocol. Request For Comments 958, Internet Directory Services, September 1985.

[12] P. V. Mockapetris. Domain names - implementation and specification. Request For Comments 1035, Internet Directory Services, November 1987.

[13] J. Moy. Ospf version 2. Request For Comments 2328, Internet Directory Services, April 1998.

[14] Midnight Networks. Automated network validation library. http://www.midnight.com/ANVL.

[15] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, April 1994.

[16] D.C. Plummer. Ethernet address resolution protocol: Or converting network protocol addresses to 48.bit ethernet address for transmission on ethernet hardware. Request For Comments 826, Internet Directory Services, November 1982.

[17] J. Postel. User datagram protocol. Request For Comments 768, Internet Directory Services, August 1980.

[18] Y. Rekhter and T. Li. Border gateway protocol. Request For Comments 1771, Internet Directory Services, March 1995.

[19] V. Jacobson S. McCanne. The bsd packet filter. a new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX Conference*, San Diego, CA, January 1993. USENIX.

[20] J. Schonwalder. Tcl extensions for network management applications. In *Proc. 3rd Tcl/Tk Workshop*, Toronto Canada, July 1995.

[21] UCB/LBNL/ViNT. Network similator 2. http://www-mash.cs.berkeley.edu/ns.

[22] S. McCanne V. Jacobson, C. Leres. Tcpdump Manual Page. Documentation for the tcpdump utility.