pktd: A Packet Capture and Injection Daemon

José María González
chema@cs.berkeley.edu

Vern Paxson
vern@icir.org

***Abstract*—**
Administrators can be highly reluctant to run foreign measurement tools on their hosts because *(a)* such tools frequently require privileged execution in order to transmit customized measurement packets and/or to passively capture network traffic, and *(b)* the administrators lack mechanisms to control the rate, duration, type, destination, and contents of traffic generated by the measurements. We present preliminary work on *pktd*, a packet capture and injection multiplexer daemon that provides controlled, fine-grained access to the network device. On systems running *pktd*, client measurement tools are not given direct access to the network device. Instead, they are obliged to request access via *pktd*. By providing administrators control over the *pktd* mechanism, they can easily and securely enforce their desired policies concerning which clients should be granted which sorts of network access capabilities. Thus, *pktd* can serve as the sole trusted, privileged entity for conducting measurements, eliminating the need for administrators to vet the individual measurement tools.

I. Introduction

Recent years have seen the developement of a number of "measurement infrastructures" [1], [2], [3], [4] in which a collection of Internet hosts support access to one or more measurement tools used for conducting measurements either between the hosts or along any Internet path stemming from one of the hosts. Some of these infrastructures, such as NIMI (National Internet Measurement Infrastructure [4]), aim to provide broad public access to the infrastructure, and also to support as many different measurement tools as possible. However, as related in [5], such open access raises some difficult problems, including how the administrator of a measurement host can control the activities of the measurement tools that others run on the host (what traffic they generate and capture), and how the administrator can safely install new measurement tools when to date such tools have frequently required privileged execution in order to transmit customized measurement packets and/or to passively capture network traffic.

The solution we envision for addressing these administrator concerns is to isolate packet capture and injection functionality into a single daemon. The administrator then requires that any measurement tools they install, or allow to execute on behalf of others, have their measurement activities funneled through the daemon, which can then provide a single point of fine-grained control.

Administrators benefit from the use of the daemon as *(a)* they have a single trusted, privileged entity in their hosts, i.e., the daemon; *(b)* the daemon is more static than the client tools, permiting deeper analysis; *(c)* the daemon provides fine-granularity access, so the administrator need grant only the minimum capabilities clients need to carry out their tasks; and *(d)* the administrator can make more efficient use of scarce resources (such as packet filter access), as the daemon can multiplex them among several clients.

Users of the measurement infrastructure, on the other hand, can hope to leverage administrator confidence in the daemon to be granted access to more hosts. At the same time, the finer granularity of the daemon may mean access to more capabilities, as administrators are more willing to provide access when it can be controlled.

In this paper we describe the design and features of such a daemon. We also present an implementation, *pktd*; its architecture; and preliminary experience obtained by its deployment in an operational environment characterized by high network load conditions.

The remainder of this paper is structured as follows. Section II presents the client-control mechanisms provided by the daemon to administrators. Section III describes the design, architecture, and preliminary implementation of the daemon. Section IV discusses the experience obtained while writing the daemon, and performance issues related to its presence between clients and the network device. Finally, Section V summarizes.

II. *pktd* Control Mechanisms

pktd is the sole trusted, privileged entity that a host providing external measurement services needs to support. It is granted full access to the network device, and hence for both security and privacy concerns must be fully trusted. *pktd* multiplexes this access among clients, providing them with different rights. Clients can access the network only by requesting measurements to *pktd*, which in turn decides whether to grant access depending on the administrator policy. A measurement might consist of *capturing* traffic using the packet filter, or *injecting* "raw" packets into the network (for example,

those with altered IP headers such as to control the TTL hop-count field).

pktd can support both passive and active measurements. Passive measurements consist of capturing packets at the host network device, without affecting the network traffic. It is therefore a non-intrusive activity, and as such the main concern for administrators is privacy (as well as some forms of security, such as protecting clear-text passwords). Active measurements, on the other hand, consist of injecting packets into the network device; this poses the security problem of ensuring that the traffic does not facilitate any form of attack, whether an *exploit* in which the injected traffic illicitly manipulates a remote service, or a *denial of service*, in which the traffic impedes access to a remote service (for example, because the volume of the measurement traffic is so large that it fully clogs the network path to a given remote server).

In addition to security and privacy, another *pktd* goal is to control network access with a granularity fine enough to match administrator concerns and user requirements at the same time. While hosts often provide mechanisms to control access to the network device, the semantics of these mechanisms are too coarse, and often entangled with other access control mechanisms. For example, any passive access to the network device in Linux platforms must be privileged; once granted, there is no further way to restrict what traffic is passively recorded.

Solaris and BPF-based [6] architectures (which include most BSD Operating Systems), on the other hand, implement network device access rights by associating the network device with a virtual file (device), where the traditional file owner/group/other read/write access holds. A client can have either full read access to the packet filter (including setting it to promiscuous mode), or none at all. There is no way to implement finer-grained policies, however, such as only allowing client access to specific types of packets and/or parts of a packet.

A related objective of the packet daemon is to provide a mechanism to enforce resource control, such as limiting the amount of traffic a client can capture or inject.

Finally, the packet daemon is intended to multiplex resources among clients. Network device multiplexing is complicated by configuration issues and operating system limitations. As an example, BPF-based platforms limit the number of processes listening to the network device to one per virtual BPF file. By multiplexing, the daemon can provide multiple clients with access to a single device.

As discussed above, administrators typically have three different concerns: privacy, security, and resource control. Two of the key design decision are: what is the right granularity to offer, and how to express it. We address these questions for each of the concerns in the remaining subsections.

A. Privacy and Security

Privacy (security) concerns are addressed by controlling the type and contents of the traffic that can captured (injected). Control is in terms of restricting (1) what types of packets can be captured or injected, and (2) how much access is provided to the contents inside the packets.

1) Traffic Type: The first mechanism provided consists of selecting which traffic can be accessed. For example, it may make sense to provide a client with access to packets arriving to or being sent from the SSH port, where all sensitive data is encrypted; but not from the Telnet port, where passwords go in the clear.

Traffic selection is based on the most common mechanism, namely *tcpdump* filter expressions [7]. The idea is that administrators can associate with every measurement client two different “framework filter expressions”: one for the capture side, and the other for the injection side. Each framework filter defines the least restrictive expression a client can request. That is, all captured traffic must match the capture filter in order to be delivered to the client, and all inject traffic must match the injection filter before being put in the wire.

This approach presents several advantages. First, it is convenient, as both administrators and clients are likely familiar with expressing *tcpdump* filters. Second, it is easy to implement. *pktd* clients submit their own requested expressions, which are automatically appended using a logical AND operation with the corresponding framework filter expression. Third, this approach provides quite fine granularity: current *tcpdump* expressions support source and destination addresses (either at datalink or network layers), network prefixes, port numbers, packet length, and arbitrary header and payload fields.

2) Traffic Contents: In parallel with restricting *pktd* clients with regards to the type of traffic they can capture/inject, a second restriction is on the extent of the packet contents that they can access. One way to restrict content access is via the amount of payload data a client can capture or set. As with *tcpdump* and the popular packet capture library *pcap* (which is used by *tcpdump*), this value is controlled by the *snaplen* (“snapshot length”) setting. For example, administrators can ensure that a client monitoring HTTP performance never accesses sensitive HTML text in the HTTP payloads by limiting *snaplen* to the sum of IP and TCP headers.

The definition of *snaplen* in *pktd* is currently independent to any other restriction, though it may be

useful in the future to combine it with the filter expressions. While this capability is not provided by the *pcap* library,[1] it is not that difficult to include in the daemon. For example, assuming all IP and TCP headers are contained in 60 bytes, an administrator could provide a client with full access to SSH traffic but not to the HTTP payloads by using the filter expression "`(port http and snaplen 60) or (port ssh and snaplen 65535)`". The combined check can be carried out during the second filtering at the daemon (see Section III-B).

However, we note that as described here, this solution is imperfect, because a *snaplen* of 60 bytes might fail to include the entire TCP/IP headers (if extensive options are used) or might instead include some of the payload (if no options are used). A potential enhancement is to permit defining *snaplen*s as a function of the network and transport layers' header lengths.

In addition, we envision cases where *snaplen*-like granularity is not enough. Different fields in the IP, TCP, and UDP headers present different privacy or security concerns. In the capturing case, addresses and ports are usually more sensitive data than, say, TTLs or fragmentation offsets. The same applies to the injection case. An administrator may be willing to permit a client to set IP fragmentation fields in order to study how they are handled by in-the-middle routers; but, on the other hand, allowing clients to set the source IP address facilitates IP spoofing, so it is less likely to be permitted. These types of concerns are not addressable just using *snaplen*s.

We envision a mechanism to support this granularity, based on processing at the daemon. Each client gets assigned two permission masks, one for the capturing case and one for the injection case. In each mask, each bit represents whether access to a specific field is allowed. By "access" we mean whether the specific field can be read, for the capturing case, or set by the client, in the injection case.

A further refinement is anonymization. In the capturing case, there are some fields the administrator may not be willing to provide for privacy concerns (IP addresses). But, at the same time it is possible to distill some information from them that, while diluting the privacy-concerning information, is still useful enough for clustering purposes. *pktd* can provide IP address anonymization, where IP addresses are scrambled while retaining the relationships between their high-order non-zero bits [8].

[1] It is, in fact, already provided by the BPF kernel module; but *pcap* does not provide an API for defining packet filters that compute it dynamically.

B. Resource Control

While restrictions discussed in the previous sections focused on privacy and security, another concern for administrators is limiting the resources a client consumes. By resources we mean principally the amount of traffic that clients are allowed to receive or send; it could also refer to processor or disk usage, which we do not consider here. For the injection case, the implementation we are considering consists of defining token bucket parameters for each client [9].

1) Sampling: Another option we are considering is to permit statistical capturing of packets. Some clients may not require access to all packets matching their filter, but only a representative fraction of them. This makes sense when the expression returns an amount of traffic large enough to overwhelm the host or the client.

One issue with providing sampling is the granularity at which it should operate. For example, keeping each captured packet with a fixed probability p implements *per-packet* sampling, which can be used to estiamte aggregate traffic effects, but does not allow much analysis of intra-connection dynamics. A different (or additional) approach would be to keep each captured packet if its connection tuple matches a given random hash; this provides *per-connection* sampling, where for any particular connection, either all of its packets are sampled, or none. More generally, the PSAMP working group of the IETF is exploring methods for expressing packet sampling [10], [11].

III. Design, Architecture, and Implementation

Figure 1 depicts the system architecture. *pktd* is composed of two different systems, the *daemon* (which is made up of three processes, `smgr`, `fmgr`, and `imgr`) and the *stub library* that clients use to interact with the daemon.

The following sections describe the main daemon functioning and parts.

A. Daemon Functioning

The *pktd* daemon accepts requests from clients on a well-known socket port on the loopback interface, and also monitors the network capture device interface. When a client process located on the same host wants to carry out a measurement, it contacts the daemon with a measurement request and some identifying information.

The daemon valides this information and then compares it with a client-specific access control list (ACL) to decide whether to service the request. If the request is accepted, the daemon carries out the corresponding measurement, forwarding the resulting data to the client.

Apart from listening to clients requesting measurements, the daemon has to monitor the capture device

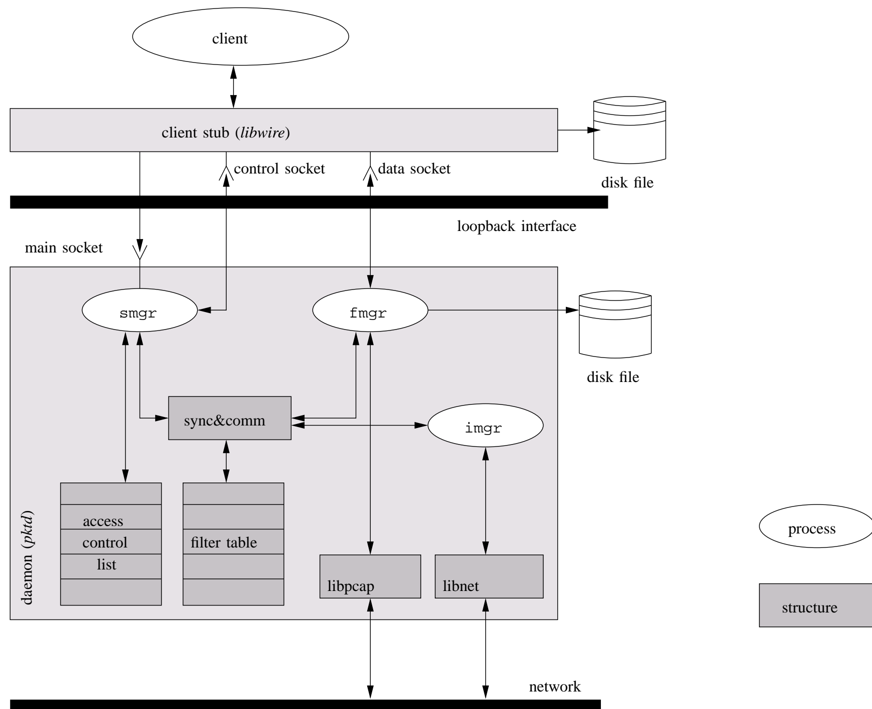


Fig. 1. *pktd* System Architecture

interface where the packets correpsonding to the measurements arrive. When it receives a packet from the network device, the daemon determines which client(s) are interested and may receive it, and accordingly forwards the packet over the corresponding local socket(s).

The daemon's third task is the careful management of packet injection.

The daemon therefore needs to monitor three event sources concurrently: First, it has to serve clients requests for capture and injection measurements at the loopback interface. Second, it has to collect packets received at the capture device interface, and forward them to the corresponding clients. Third, it has to carry out high-precision timing for the packets it has decided to accept for injection.

In order to prevent one event source from hindering service to another, the daemon is composed of three processes running concurrently: *filter manager* (`fmgr`), *socket manager* (`smgr`), and *injection manager* (`imgr`), respectively.

B. Filter Manager

The filter manager (`fmgr`) is in charge of operating the packet capture device. This means setting the correct device filter and the number of bytes the device has to capture per packet (the *device snaplen*). It also forwards packets matching their filters to corresponding clients.

Client requests are composed of a client filter and the number of bytes per packet they are interested on (the *client snaplen*). When the filter manager receives requests from more than one client, it sets the device filter to an OR'ed juxtaposition of all client filters, and the device *snaplen* to the largest client *snaplen*.

When a new packet is captured by the device filter, `fmgr` determines which client filters match the packet. For each matching client, the packet is trimmed according to the client *snaplen*, and then forwarded.

There is a performance problem with using the same capture device for all clients. The *pcap* library does not provide a way to reconfigure the *snaplen* of an already-open device, so it has to be set when the device is first opened. Using a large, fixed *snaplen* that covers all possible client *snaplen*s (i.e., the maximum transmission unit of the underlying network) means the capture device will capture more data than needed from the network device, therefore degrading the capture performance.

A first approach is to emulate a device *snaplen* change call by closing and immediately reopening the device. But in order to ensure no packets are lost during the transition, the daemon must open a new capture device with the new *snaplen*, synchronize both devices, and then switch from the old device to the new one. Synchronizing packets, though, is cumbersome, as *pcap* does not ensure that timestamps for the same packet as seen by two

different capture devices are the same.[2]

But even if the daemon could seamlessly change the device *snaplen*, we would still have a performance problem. One client requesting a large *snaplen* affects the performance of all of the remaining clients, even if its filter rarely matches. Our solution is for *pktd* to keep *two* open devices, each with a fixed *snaplen*. One device has its *snaplen* set to the maximum *snaplen* possible (*slow device*), while the other has it set to a quantity just sufficient to capture IP and transport headers (*fast device*). Clients are served by the slow or the fast device depending on their requested *snaplen*. The latter is used to attend clients requiring high-speed packet capture, while the former ensures all clients can be served.

A two-device solution matches the two types of clients we expect for *pktd*: a bimodal use pattern consisting of clients interested in measurements focused on datalink, network, and transport layers, and clients interested in upper layers. Therefore, the fast device *snaplen* is set to the sum of the link, network (IP), and largest transport protocol header size. Using the common assumption that IP options are virtually never used, we can set this value to 80 bytes (20 for the IP header and 60 for the maximum TCP header with options) plus the link layer length. [3]

A two-device solution with fixed *snaplen*s avoids the hassle of synchronizing the capture devices, adapts better to fast-changing environments that we expect common during some types of *pktd* use (such as SCNM [12]), and is easier to implement. The main drawback is that it requires keeping two capture devices open. For hosts where the daemon is not able to open two devices at the same time, *pktd* provides a fallback of just using the slow filter.

C. Socket Manager

The second *pktd* process is the socket manager (`smgr`). `smgr` follows a typical client-server model, listening for service requests from clients. The interaction is controlled by an access control list (ACL), which enforces the policy to be applied to each client.

`smgr` is bound to the loopback interface, so only requests from client processes executing on the local host can be received. While it is easy to extend `smgr` to accept requests from processes located in other hosts, we refrain from doing so in order to limit security concerns. Accepting only local host clients also permits inexpensively implementing the client access control list based on user IDs, which wouldn't work for remote clients.

D. Synchronization and Communication

The three processes need a means to communicate information about the state of the filter and ACL tables. (See below for a description of the filter table.) Process communication can be performed using either shared memory or loopback-interface sockets. In the first case, the filter table is memory mapped using *mmap*, and access is synchronized using semaphores. Changes in the filter table are reflected automatically in the other process.

Unfortunately, some systems where we are interested in running *pktd* do not support semaphores. In this case the daemon can be compiled to use sockets as the means to perform IPC. In this case, one of the processes is set as the responsible for the table, and it has to attend petitions from the others to perform changes. In the case of the filter table, the original copy resides in the `fmgr`, as this is the most thorough user.

The *filter table* stores information regarding the state of all active measurements. The filter table is updated every time clients request or finish measurements.

Both `fmgr` and `smgr` need access to the table filter. `fmgr` needs it to manage the capture device, and `smgr` to attend client requests. They need to have a consistent view of the measurement list, so any modification in one's copy must trigger changes in the other.

E. Off-line Mode and Checkpointing

pktd may also be used to record traffic for off-line analysis, using the *daemon dump* functionality. This consists of *pktd* clients instructing the daemon to dump the client's packets to a file instead of sending them over the connection between the client and *pktd*. This way, a client can request a long-term measurement, and get the file name from the daemon. Once the measurement has finished, the client can fetch the *tcpdump* file.

File naming poses a security issue. In order to avoid client attacks based on clobbering system files, clients are not permitted to select the file name; it is instead selected by the daemon.

Another problem with daemon dump is that disk consumption may grow without limit. *pktd* supports *checkpointing*, meaning that as trace files reach particular thresholds, *pktd* closes the current file and opens a new one. In addition, *pktd* has a notion of how much disk it has the right to use, and deletes the oldest files as that space becomes exhausted.

Checkpointing can be requested using three different metrics: time, packet count, or disk space. Measurement

[2]Experiments we conducted on a FreeBSD 4.7 host found that reported timestamps could, under some circumstances, differ between 4 and 7 microseconds for the same packet in two concurrently open devices.

[3]Ideally, *pcap* would support richer semantics in the *snaplen* choice. Instead of 80 bytes, we would like to select the *snaplen* to be exactly the datalink, IP, and TCP layers. There is actually enough flexibility in the BPF language to support doing so, but no way to access it from *pcap* and the *tcpdump* filtering language it uses.

limits can be expressed as "trace for the next T seconds," "capture at most N packets," or "generate a file no larger than B bytes," respectively.

F. libwire *Library Stub*

Clients access the daemon using a library stub called *libwire*. *libwire* provides clients with a simplified version of the *pcap* programming API. Clients also describe measurement filters by using *tcpdump*-like filter expressions.

Client tools are linked with *libwire*, which provides access to the daemon.

Table I describes the library stub application programming interface.

G. Injection Manager

For some types of measurements, it is essential to inject customized packets into the network. While some software packages standardize raw access to the network device, we again want to provide fine-grained policy decisions to the box owner. For example, it may be OK for a client to inject a well-formed ICMP packet addressed to a computer whose distance to the measurement box is being calculated. On the other hand, full raw access to the network device is a capability few system owners are willing to permit to most clients; it can be used for all sorts of attacks.

Packet injection is carried out by the injection manager (`imgr`). `imgr` is actually implemented in two different ways: using raw sockets, and writing directly to the link-layer interface. Differences are related to the functionality provided and the simplicity of the implementation. The current implementation can be configured at compile time to use any of them.

The first case obliges the daemon to be run as root in Linux, FreeBSD, or Solaris. Direct writing to the link-layer interface can be implemented in FreeBSD just by providing the daemon with write access to the BPF device.

We envision injection to work as follows: when a client wants to inject a packet, it provides the packet to the daemon along with the exact time when it wants it to be injected. Precise timing of the transmission can be implemented by combining gross time scheduling (via *usleep*) and then, for the final milliseconds, busy-waiting (the approach used by the "`zing`" utility distributed with NIMI).

The daemon would return a unique tag to the client for each injection request. Once `imgr` eventually sends the message, it will then forward to the client the exact timestamp when it was sent.

We are very interesting in supporting high-precision measurement. As such, we would like to give as much precision as possible, if not for measurements across different hosts (the correctness of the any time information distilled from measurement taken in two hosts is only valid if they are synchronized tightly enough), at least for measurements in the same host.

On the capture side, the best timestamps we can get for a captured packet are those provided by BPF. On the injection side, we can (a) listen to our own packet and accept the timestamp returned from BPF, and (b) take a timestamp when we do send the packet. In the second case, the idea is to take a timestamp before sending the packet, and after transmission returns, and distill some value in the middle (or perhaps simply report both).

We will also need to do some work to assess scheduling precision across several platforms.

Clients are also allowed to define the initial bytes of the packet: for example, the network and transport headers, and perhaps some additional initial payload, plus a pattern to compose the rest of the packet. The `imgr` would check which fields the client can select, and overwrite the rest.

Given the potential power of packet injection, not all host administrators will want to make it available. Thus, the packet injection capability can be removed from *pktd* at compile time. Clients know whether the daemon supports injection by checking the results of the `wire_init` API call.

H. Compression

When capturing very high volume traces, the overhead of copying the packets—even if only headers—can become significant. Thus, as part of the communication between `fmgr` and the client, we implemented *header compression* to reduce the data volume we need to transmit.

In general, our compression implementation follows the approach used by CSLIP guidelines [13], but with some differences. First, CSLIP relies on the underlying link layer for conveying packet size information, whereas our compression must be fully self-contained. Second, we need to include timestamps as well as the packet headers.

Third, CSLIP was designed to operate primarily in environments where space was at a greater premium than computation cycles, so squeezing every last bit of compression, at the cost of some non-aligned operations, makes sense for it. For *pktd*, on the other hand, we're more concerned with saving cycles than with squeezing every bit. Accordingly, for our compression we rely on byte operations instead of bit-wise ones.

Finally, unlike with CSLIP we can consider forms of "lossy" compression, since for some forms of measurement (those with *snaplen* shorter than the packet payload, or for which the administrator prohibits access

API call	Parameter	Explanation
`wire_init`		*connect to the daemon*
	filter	string, filter to be installed
	snaplen	integer, amount of each packet to capture
	mode	flag, connection mode (includes disk dump, buffering, and compression)
	cp	`struct`, checkpointing request
	co	`struct`, compression request
	write_file	string, file where trace should be written
`wire_done`		*close and cleanup associated state*
	pdd	*pktd* connection descriptor
	ps	*pktd* capture statistics
`wire_stats`		*get* pktd *statistics*
	pdd	*pktd* connection descriptor
	ps	*pktd* capture statistics
`wire_setfilter`		*request a new filter*
	pdd	*pktd* connection descriptor
	filter	string, new *tcpdump* filter to be installed
	cp	`struct`, new checkpointing request
`wire_checkpoint`		*request the daemon to checkpoint*
	pdd	*pktd* connection descriptor
`wire_flush`		*flush data in daemon-buffered connections*
	pdd	*pktd* connection descriptor
`wire_activity`		*receive a packet from the daemon*
	pdd	*pktd* connection descriptor
	cb	callback for each captured packet
	user_data	user-defined data to pass to the callback
`wire_inject`		*request a packet to be transmitted by the daemon*
	pdd	*pktd* connection descriptor
	ip	binary, pointer to the IP header of the packet to be injected

TABLE I
MAIN API CALLS FOR *libwire*

to particular header fields) we do not have to represent the full semantics of the packet. For example, if the client is not allowed to see the IP identification field, why waste time and bandwidth in compressing and sending it? Thus, we have opted for a cleaner separation of IP, TCP, and UDP headers, in order to accommodate "lossy" trace compression.

pktd's compression exhibits quite good performance. See See Section IV for a discussion.

IV. EXPERIENCE AND PERFORMANCE RESULTS

pktd is implemented in 12,000 lines of C code, on top of *pcap* and *libnet* [14]. It has been ported to Linux, Solaris, and FreeBSD. *pktd* is distributed under a BSD-style license, and is provided by the authors on-request.

Much of our initial evaluation of *pktd* is in the context of a project (SCNM; see below) requiring very high-speed (Gigabit Ethernet) capture. Therefore, a main focus for the daemon has been to tune it to avoid packet loss in the presence of very high volume streams as much as possible. Tuning led us to thinning the daemon as much as possible, especially the code executed per every packet. This part of the code is critical in performance terms; as we will see below, it influenced several further design decisions.

Our performance analysis was mainly done using a high bitrate "test stream" consisting of an 832 Mbps (159 MB in 1.53 seconds) UDP flow, which was comprised of 113,120 large, heavily fragmented packets.

One of the problems of tuning the daemon was determining what constituted a good target—i.e., a good tool against which to compare. On our evaluation platform (FreeBSD 4.4), plain *tcpdump* loses packets when capturing the test stream. The problem is not simply the

volume of the stream: if we capture just the headers, we still typically lose 0.2 % of the packets. Even a relatively small drop rate such as this can be a significant difficulty when trying to diagnose problems like which node along the full path is losing packets. If we try to capture more data per packet, the drop rate goes up. For example, *tcpdump* loses 10 % of the packets when *snaplen* is 150 bytes.

The first tuning consideration is the influence of *pcap*'s setting of the BPF kernel buffer size. Plain *pcap* sets the kernel buffer size to 32 KB, which for a high volume stream leads to a great number of context switches as captured packets are transferred into the user-space, especially if we capture more than just headers (and hence the buffer fills up faster). Increasing the buffer size can help a great deal when capturing full packets. However, surprisingly a larger kernel buffer can actually *decrease* performance when capturing small packets (why remains a puzzle—perhaps due to L1 cache effects), as illustrated below.

A second consideration is the use of *interrupt coalescence* in the NIC [15], [16]. In order to achieve high-performance capture, it is crucial to avoid the network card generating a new interrupt for each arriving packet. Our test stream, for example, would generate one interrupt every 14 μs—definitely an extreme processing burden on the kernel.

Some network cards provide an interrupt moderation feature, also known as interrupt coalescence, which bundles several packets into a single interrupt. The idea is that the NIC, after receiving a packet, does not automatically generate an interrupt to request the CPU transferring the data to memory. Instead, the interrupt is delayed for up to a given amount of time (the *interrupt moderation period*) in hopes of other packets arriving in the meantime and being served by the same interrupt.

Too large an interrupt moderation period can lead to large delays in packet processing. This is a problem when the captured traffic is interactive, i.e., when there is the need of answering it in real-time (for example, by a reactive intrusion detection system). However, this is not the case for *pktd*. Thus, all of our experiments were run using a 3 ms interrupt coalescence (compared to the 200 μs default value in FreeBSD 4.7).

Another important tuning element is careful buffer management. In order to deliver high volumes of packet headers to its clients, *pktd* naturally needs to buffer the headers before sending them in large units. Initially, we used the standard C *stdio* library. We then found, however, that (at least under FreeBSD) the performance of the standard C *stdio* library in the specific case of doing small writes is slightly worse than what we could get using our own user-level buffering. Small writes (packet headers) are the main scenario where we have tested *pktd*, so we built our own user-level buffer management library, *lstdio*.

Figure 2 shows the performance obtained as a function of the write size for unbuffered writing (*write*), the C standard library using the default 8 KB buffer size (*fwrite*), and the user-level library we wrote for *pktd* with the same 8 KB buffer size (*lfwrite*). We see that the performance difference between the standard C library and the user-level buffering is very small, only 10% for the case of 64-byte writes. But this small difference can be important in the high-load conditions; as we will see, it accounts for *pktd* slightly outperforming *tcpdump*.

Figure 3 shows the result of applying all of the discussed ingredients. We have run plain *tcpdump*, *tcpdump* changing the kernel buffer size to 1 MB, and *pktd* also using 1 MB of kernel buffer space. All experiments were carried out in the same way: an unloaded capture host with a GigEthernet NIC sees the test stream and tries to capture as much traffic as possible. The top drawing shows the percentage of packets captured, and the bottom one the total number of bytes captured.

There are several interesting aspects in the figure. First, we can see how the kernel buffer size affects *tcpdump* performance. A small buffer size is better when capturing few bytes per packet, but performance degrades rapidly when increasing it.

pktd does not lose traffic until you pass the 94-byte *snaplen* milestone. That number is a consequence of the two-device (*fast* vs. *slow*, per Section III-B) architecture in the daemon. If *snaplen* is 80 bytes (20 for the IP header and 60 for the maximum TCP header with options) plus the link layer length (14 bytes for the Ethernet header) or smaller, then the *fast* device is used (with 94 bytes of *snaplen*). Once the capture size surpasses the 94 byte limit, *snaplen* is set to the maximum possible amount (the *slow* device), so performance degrades very quickly.

As mentioned in Section III-H, *pktd* supports compression of packet headers when conveying them from `fmgr` to the measurement client. We find that the compression works quite well: *pktd* carries out lossless compression of UDP headers with just 7 bytes/packet (including timestamps), and TCP headers in only 15 bytes/packet, including timestamps and TCP options. The UDP traces are those used for the test stream: synthetic, fat traffic (heavily-fragmented UDP packets with an average length of 1500 bytes) corresponding to a single connection. The TCP traces are from measured FTP traffic, with an average 4.16 bytes of options per packet.

Compression is carried out with no noticeable performance loss. Using a 68 byte *snaplen*, we can support multiple concurrent *pktd* client capturing the test stream

PSfrag replacements

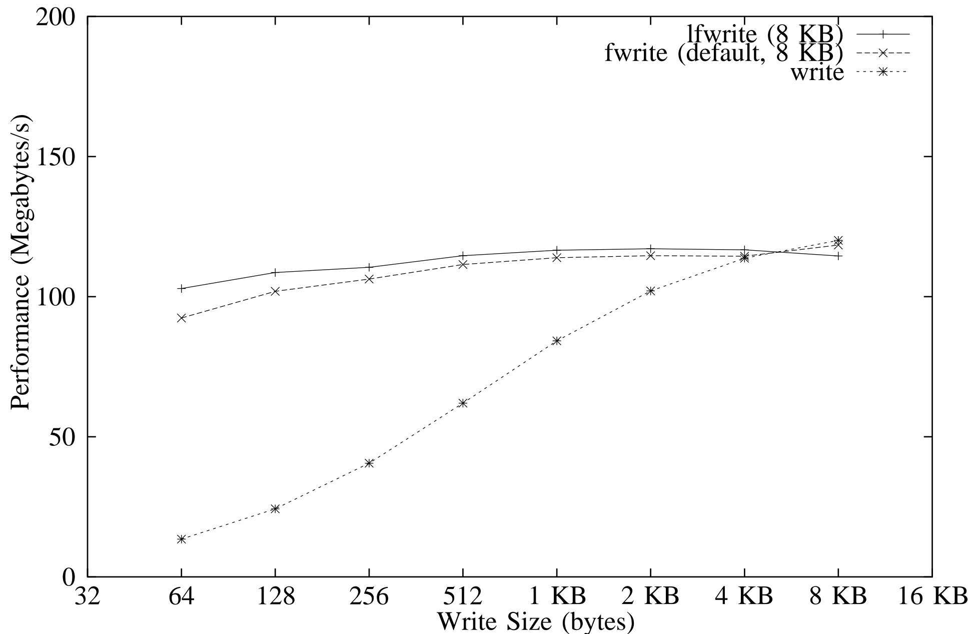


Fig. 2. User-Level Write Buffer Performance

with no packet losses (we've tested up to a dozen). This is a consequence of *pktd* spending less memory bandwidth when transmitting compressed copies of the headers to the clients, a savings that more than offsets the extra cycles required for compression and decompression

V. Conclusions

We have designed and implemented *pktd*, a packet capture/injection daemon that provides policy-controlled, fine-grained access to the network device. Our preliminary measurements have found that, once we tuned the implementation, it is able to capture high-volume streams of traffic without significant performance problems.

pktd is a helpful tool for implementing safe measurement techniques, where "safe" means that tools needing packet capture or injection can only capture or source traffic that conforms with the local administrative policy, and themselves require no special privileges.

pktd is part of the NIMI project infrastructure, and is also being used by the Self-Configuring Network Monitor (SCNM) Project [4] at several U.S. National Laboratories [12]. Source code is available on request.

VI. Acknowledgments

We gratefully acknowledge Deb Agarwal, Brian Tierney, and Jin Guojun for all of their help during *pktd* development. They have been our main users, and as such provided invaluable comments and suggestions. Thanks also to Andrew Adams for helpful discussions.

This work was partially supported by the Spanish *Ministerio de Ciencia y Tecnología*'s ICSI fellowship, and the NSF National Middleware Initiative, Award 0222846.

[4]http://www-itg.lbl.gov/Net-Mon/

References

[1] G. Almes *et al.*, "Surveyor Home Page, Tools, and Infrastructure," 1997. [Online]. Available: http://io.advanced.org/surveyor

[2] C. Labovitz *et al.*, "The Internet Performance and Analysis Project," 1997. [Online]. Available: http://www.merit.edu/ipma

[3] H. Werner-Braun *et al.*, "Active Measurements, Tools, and Infrastructure," 1998. [Online]. Available: http://amp.nlanr.net

[4] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis, "An architecture for large-scale internet measurement," *IEEE Communications*, vol. 36, no. 8, pp. 48–54, August 1998.

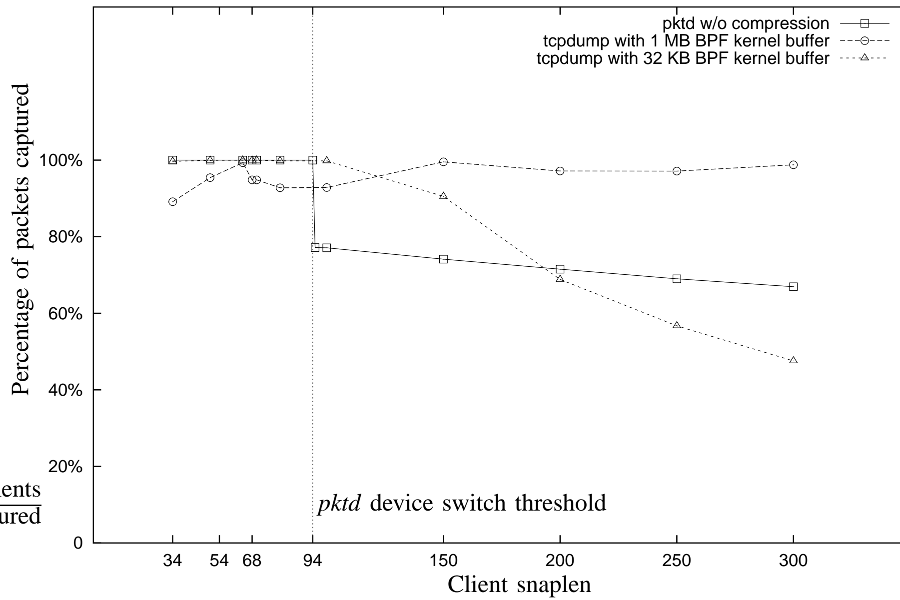
[5] V. Paxson, A. Adams, and M. Mathis, "Experiences with NIMI," in *Proceedings of Passive and Active Measurement*, 2000. [Online]. Available: http://citeseer.nj.nec.com/paxson00experiences.html

[6] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *USENIX Winter*, 1993, pp. 259–270. [Online]. Available: http://citeseer.nj.nec.com/mccanne92bsd.html

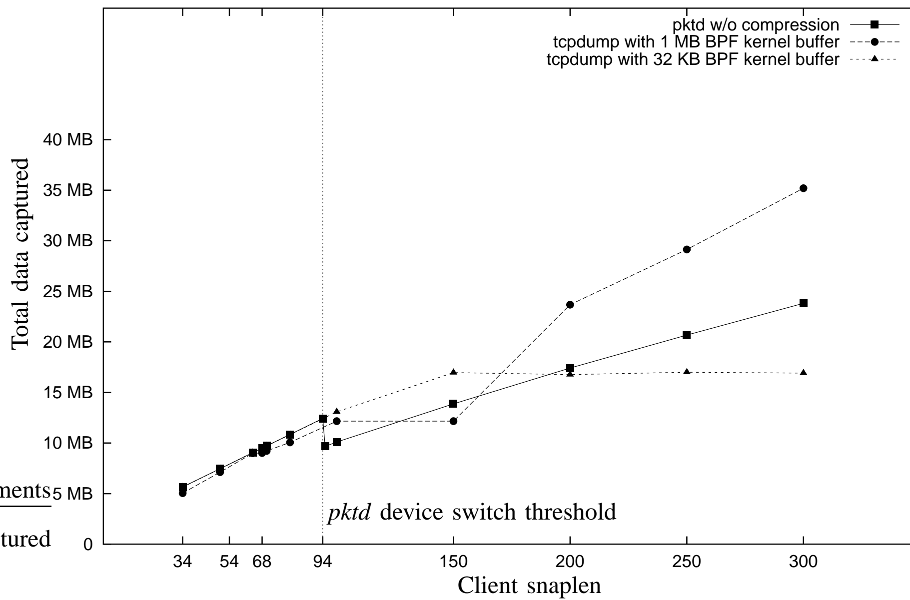
[7] V. Jacobson, C. Leres, and S. McCanne, "tcpdump—dump traffic on a network. UNIX man page," 1993. [Online]. Available: http://www.tcpdump.org

[8] G. Minshall, "TCPdpriv command manual," 1996. [Online]. Available: http://ita.ee.lbl.gov

[9] J. Turner, "New directions in communication (or which way in the information age)," *IEEE Communications Magazine*, vol. 24, no. 8–15, 1986. [Online]. Available: http://citeseer.nj.nec.com/context/311305/0



PSfrag replacements
Total data captured



PSfrag replacements
Percentage of packets captured

Fig. 3. *pktd* vs. *tcpdump* Performance for an 800 Mbps Stream

[10] N. Duffield *et al.*, "A Framework for Passive Packet Measurement (work in progress)," 2002. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-psamp-framework-01.txt

[11] P. S. (psamp) Charter, "Packet sampling (psamp)," 2002. [Online]. Available: http://www.ietf.org/html.charters/psamp-charter.html

[12] D. Agarwal, J. González, G. Jin, and B. Tierney, "An Infrastructure for Passive Network Monitoring of Application Data Streams," in *Proceedings of Passive and Active Measurement*, 2003. [Online]. Available: http://www-itg.lbl.gov/Net-Mon

[13] V. Jacobson, "Compressing TCP/IP headers for low-speed serial links," 1990. [Online]. Available: http://citeseer.nj.nec.com/ncontextsummary/60581/0

[14] M. Schiffman, "The libnet reference manual," 1999. [Online]. Available: http://www.packetfactory.net/libnet/manual

[15] J. Guong, "Personal communication," 2002.

[16] C. Leres, "SCNM FreeBSD mods," 2002. [Online]. Available: http://www-didc.lbl.gov/SCNM/FreeBSD_mods.html