Passive & Active Measurement Workshop (PAM2002),
Fort Collins, Colorado, March 2002. LA-UR 02-0809

TICKETing High-Speed Traffic with Commodity Hardware and Software

Eric Weigle† and Wu-chun Feng†‡
{ehw, feng}@lanl.gov

†Computer & Computational Sciences Division
Los Alamos National Laboratory
Los Alamos, NM 87545

‡ Department of Computer & Information Science
Ohio State University
Columbus, OH 43210

Abstract

While tcpdump is an invaluable monitoring tool that has held up remarkably well for over a decade, it is showing its age. Network speeds have recently outstripped the ability of 'stock' tcpdump running on commodity hardware to keep up with the network, rendering it incapable of monitoring traffic at gigabit-per-second (Gbps) speeds. Tests over Gigabit Ethernet showed that tcpdump could monitor and record traffic at speeds no greater than 250 Mbps with O(ms) time granularity.

To achieve monitoring at Gbps speeds and O(ns) time granularity with commodity parts, we present TICKET – the Traffic Information-Collecting Kernel with Exact Timing. TICKET combines efficient commodity-based hardware and software in an architecture that hides disk latency and bandwidth.

Keywords: TICKET, tcpdump, libpcap, traffic collection, high-performance networking, passive monitoring.

1. Introduction

The TICKET architecture makes use of two or more commodity PCs. The first PC captures packets in real time, collects relevant data from the packets, and forwards information gathered to one or more additional PCs. These then save, display, or continue processing the data. The inherent distribution of tasks among multiple machines allows TICKET to scale where other approaches (such as tcpdump/libpcap or RMON devices) can not.

Traffic monitors that rely on tcpdump/libpcap-style processing [3, 4] gain ease of use at the cost of performance. Design decisions, such as forcing all collection, analysis, and display to be performed on a single machine, as well as implementation decisions, such as single-packet copies from kernel level to user level, mean that tcpdump and libpcap cannot scale to today's Gbps network speeds. Likewise, while traffic monitors complying with the RMON specification [10] have more potential to scale, actual implementations from prominent companies fail to reach this potential.

Thus, we present TICKET – the Traffic Information Collecting Kernel with Exact Timing. TICKET is a passive monitor that combines efficient, commodity-based software with a commodity-based hardware architecture that hides disk latency and bandwidth. TICKET provides a scalable, high-performance methodology for collecting high resolution information about all traffic traversing a single network link. TICKET and the information collected by TICKET can then be used to improve network protocols, provide insight into how to allocate network resources and improve network design, enable fast detection and correction of network problems, or enhance cyber-security.

This paper first presents background information on computing trends with their relevant implications. We then discuss traffic collection infrastructure and other related work. We continue with our methodology, how it is unique, and how it has been implemented. Next we consider some experiments and practical results using the system, before concluding with a final analysis.

1.1. Computing Trends

Network speeds have been increasing at an incredible rate, doubling every 3-12 months [2]. This is even faster than one version of "Moore's Law" which states that processing power doubles every 18-24 months; an observation made in 1965 which is still remarkably accurate [6]. Moreover, although disk sizes have been increasing dramatically, disk and bus bandwidth have been increasing almost linearly; much more slowly than the exponential growth in other areas. In short, the disparity between network, CPU, and disk speeds will continue to increase problematically.

For example, consider one point in time among these trends: today. Table 1 gives the results leading-edge researchers have shown for various computer components, and the results that are generally considered "commodity".

	Research	Commodity
Network	6.4Tbps	1.0Gbps
Processor	64-bit/4.0GHz	32-bit/1.0GHz
Memory	64-bit/333MHz	64-bit/133MHz
Accessory	64-bit/133MHz	32-bit/33MHz
Disk	160MBps	20MBps
Network	6400.0 Gbps	1.0 Gbps
Processor	256.0 Gbps	32.0 Gbps
Memory	21.3 Gbps	8.5 Gbps
Accessory	8.5 Gbps	1.1 Gbps
Disk	1.3 Gbps	0.2 Gbps

Table 1. Raw and Converted Speeds of Research and Commodity Components

The units in this table are Terabits per Second (Tbps, 10^{12} bits per second), Gigabits per second (Gbps, 10^9 bits per second), Gigahertz (GHz, 10^9 cycles per second), and Megahertz (MHz, 10^6 cycles per second). The upper half of the table has given values in their common units, while the lower half converts these values to Gigabits per second for approximate comparison. Note that while these conversions are somewhat naive, they give the benefit of the doubt entirely to the opposing point. Observing *effective* speeds, accounting for bus arbitration, cache coherency, and so forth decreases the values for Processor, Memory, Accessory, and Disk, with Network speeds remaining nearly constant.

The research values were derived as follows: Nortel Networks (among others) demonstrated 6.4 Tbps as far back as 1999, while super-cooled Intel Itanium processors, PC-166 memory, the PCI-X interface, and the Wide Ultra3 SCSI disk interface complete the table.

1.2. Implications for the Future

Predicting the future is always risky, yet most analysts agree that there are enough new manufacturing methods, designs, and physics "in the pipe" that these trends will continue for some years to come. Of critical importance is the fact that networks are becoming optical while computers are still fully electronic.

The implications of these trends are critically important. What we wish to point out (and that Table 1 should make clear) is *commodity hardware and conventional methods will not suffice to capture traffic in the future*. As research technology becomes commodity, tcpdump/libpcap style processing [3, 4], will have neither the requisite processing power, nor the disk bandwidth available to handle fully saturated network links. Any such single-machine design, no matter how well implemented, will soon be unable to "keep up" with the network.

Consider another example: field tests of tcpdump/libpcap running on a 400-MHz Pentium III Linux machine over a Gigabit Ethernet (GigE) backbone showed that tcpdump could monitor traffic at speeds no more than 250 Mbps with O(ms) time granularity. At times of high network utilization ($\approx$85%), over half of the packets were lost because it could not handle the load.

Addressing disk and accessory bandwidth by using RAID systems [8] as network-attached storage (NAS), and the memory integrated network interface (MINI) paradigm [5] is possible, but both are still somewhat expensive. Memory bandwidth issues have been addressed by technology such as RAMBUS or simple increases of processor cache size, but these techniques have failed to live up to their potential. We have tried using these approaches, in combination with highly tuned commodity-based monitors and even a prominent commercial monitor (Section 1.3). Unfortunately, we found them expensive and still lacking – susceptible to the same trends shown above. We realized that there was a need for a new, specialized monitoring methodology. This is what TICKET provides.

1.3. Related Work

We have implied that some other work has limitations that TICKET addresses, here we provide more detail. Most of these limitations are due to trade-offs where existing solutions have made different choices than TICKET. Table 2 summarizes prominent features, and discussion follows. We then discuss other approaches including tcpdump/libpcap and RMON devices in more detail.

Other Approaches	TICKET
Not scalable	Scalable
Not very efficient	Very efficient
Very easy to use	More difficult to use
O(us) resolution timers	O(ns) resolution timers
Some are portable	Not portable
Some not real-time	Can perform in real-time
Some are unreliable	Highly reliable
Brute-force capture	Parsing capture
Free to very expensive	Free to inexpensive

Table 2. Features of TICKET and Other Approaches

Scalability refers to forcing use of a single machine or processor rather than allowing use of SMP or cluster systems. On a single machine, the collection task can require so much processing power that too little remains to actually *do* anything useful with the data.

Efficiency, ease of use, timer resolution, portability, and real-time factors resolve to a user level versus kernel level trade-off. Most other approaches are user level applications tied to their operating system's performance and depend upon system calls such as `gettimeofday()`. TICKET is a dedicated operating system (OS) that runs at kernel level and uses lower-level hardware calls such as `rdtsc` (read-time stamp counter).

Reliability refers to issues with many implementations that fail to keep up with network speeds, fail to maintain a count of dropped packets, crash or hang during use.

Brute force applications take a a fixed, predetermined "capture size" from each packet, while TICKET parses packet headers on-the-fly and captures only data of interest.

TICKET is cheap, in terms of cost. Because there have been no other options, many vendors offer custom hardware solutions (RMON or otherwise) with proprietary software at an exorbitant cost; some up to $200,000 or more! TICKET provides options. Hardware to run TICKET can be put together with commodity parts for less than $2,000. The TICKET software will be available for free.

Tcpdump is an invaluable, easy-to-use, portable, free tool for network administrators. It is designed as a user interface application relying upon functionality contained in the lower-level libpcap library, which has also been successfully used with other applications such as CoralReef [1]. Unfortunately, by nature tcpdump and libpcap have limitations – due to decisions made concerning the trade-offs listed above. In particular, libpcap executes on a single machine, using system calls to perform timestamps, performs brute-force capture, and can be unreliable.

Furthermore, libpcap suffers from efficiency problems pandemic to implementations of traffic collection at user level. They must ask the operating system to perform the required packet copy in the network stack (for transparency), which can double the time required to process a packet. The exact method used by libpcap and other tools varies by operating system, but always involves a context switch into kernel mode and a copy of memory from the kernel to the user level library. This "call-and-copy" approach is repeated for every packet observed in Linux, while other implementations use a ring buffer in an attempt to amortize costs over multiple packets. At high network speeds, the overhead of copying each individual packet between kernel and user space becomes excessive enough that as much as 50% of the aggregate network traffic is dropped when using tcpdump/libpcap over a GigE link.

RMON devices [10] contain some traffic-collection functionality as well as some management functionality; that is, they provide a superset of the functionality of tcpdump/libpcap but work in much the same way. However, our RMON device from a prominent company suffers from several problems. Although the management software provides a nice interface to the hardware RMON device, it also introduces substantial overhead that limits the fidelity of packet timestamps to O(seconds); this fidelity is a thousand times worse than tcpdump and nearly a billion times worse than TICKET. In addition, the packet-capture mode of the RMON device often silently drops packets; the data-transfer mode of the RMON requires an active polling mechanism from another host to pull data across; and the RMON devices themselves hang or crash often, e.g., every 36-72 hours.

1.4. Required Infrastructure

To collect traffic, we require a network interface upon which a copy of all relevant network traffic is available. This can be done using network stack operations, port mirroring, or a tap mechanism.

Network stack operations performed by the operating system provide a copy of data to the libpcap program running on a given host. Libpcap may perform additional processing before passing it on to tcpdump for display.

Port or interface mirroring is a technique by which the traffic from one or more interfaces on a network switch (the mirrored interfaces) is copied to another port (the mirroring interface). In theory, this provides a mechanism to transparently observe traffic passing over the mirrored interfaces by observing traffic over the mirroring interface. In practice, this mechanism has problems.

One problem is that most implementations cause significant performance degradation for the switch; packet timing is skewed and many packets are dropped due to poor implementation of the required copy. Another problem is that nearly all today's connections are full duplex: mirroring a two-way stream onto a one-way (outgoing) port means that we have a potential 50% loss rate. In this case two mirroring interfaces are required, one each for incoming and outgoing traffic.

A tap mechanism is a a piece of hardware that takes a single network input and duplicates it to transparently produce two identical outputs. This can be thought of as a splitter or a switch performing half-duplex port mirroring. This hardware works at the physical level (electrons or photons "on the wire") by splitting a physical signal and possibly enhancing it.

2. Methodology

TICKET is split into two parts, running on separate hosts. First is a highly efficient dedicated operating system (OS) for initial traffic collection, and second is a set of user level tools and scripts to further process, save, or display the collected traffic. We discuss each in turn and then compare this approach with others.

Using kernel code running as close to the hardware as possible maximizes efficiency, scalability, and performance in critical areas. As a dedicated system, it avoids extraneous services competing for resources and decreasing security. Furthermore, running entirely in kernel space means that no context switches are required, and buffer copying is kept to a minimum, e.g., data no longer has to be copied from kernel space to user space. TICKET utilizes the cycle-counter register to provide high-fidelity timestamps with accuracy and precision thousands of times better than implementations of other approaches.

Using user level code maximizes usability and expressive power. Less time-critical tasks such as saving to disk are performed by this set of programs and scripts. The ability of the kernel level code to stripe across multiple user level machines makes efficiency less of an issue and enables highly complex analysis in real-time using clusters of machines.

2.1. Simplest TICKET Configuration

The simplest configuration is just a TICKET kernel machine with one interface connected to a network tap of some sort, and another interface connected to a second machine running a user level program to print data to the screen. This would be comparable to a single machine running tcpdump/libpcap.

The TICKET architecture is shown in more detail in Figure 1. This figure shows a logical view of the hardware configuration and how processing is shared among multiple hosts. TICKET first collects traffic from the network link (A) via a network tap (B). The network interface card (D) then collects traffic from the tapped network link (C), passes the data over the peripheral PCI bus (E) to the main CPU (F). The main CPU simply gathers the appropriate packet headers and passes them back over the peripheral bus (G) and network interface card (H) and onto a dedicated network link (I) to one or more application boxes (J) for display and/or storage. If the network interface card (NIC) has an on-board processor, this can be utilized to provide even better performance (see Section 3 for details).

The architecture of monitors based on libpcap is a direct contrast to the scalable infrastructure used by TICKET. As Figure 2 illustrates, tcpdump-based monitors typically run on a single bottleneck host (i.e., the "tcpdump box"). The monitor gathers network traffic data from the network link (A) via a network interface card (B) in promiscuous mode. This data then crosses the peripheral bus (C) before entering main memory and being processed by the CPU (D). Finally, the data crosses another peripheral bus (E) to be written to disk or a display device. Generally, there is little spare processing power for data analysis.

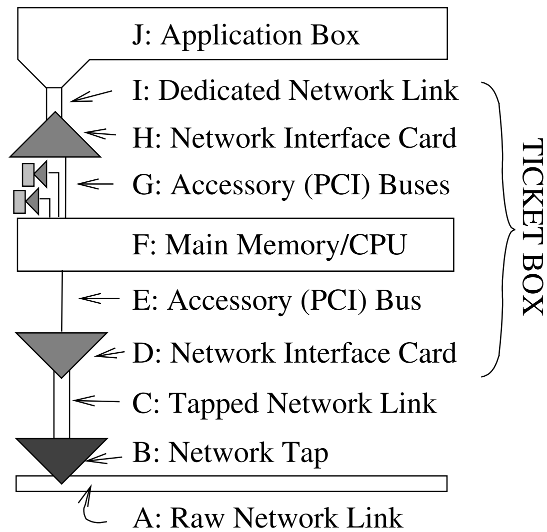

Figure 1. TICKET Hardware Configuration

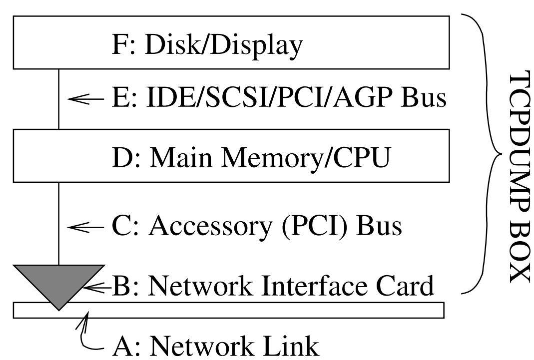

Figure 2. Tcpdump/libpcap Hardware Configuration

With the adoption of a dedicated OS for TICKET, the display and storage of data traffic become the bottlenecks in the traffic collection rather than the first-level collection and analysis (see Table 1). However, to address those bottlenecks, we architect our TICKET machine to have multiple, independent PCI busses and a Northbridge/Southbridge structure capable of operating them concurrently at full speed. When this is true, we can effectively split the traffic with no loss – hiding disk bandwidth and latency.

2.2. Alternate Configuration One

The simplest configuration scales well beyond current techniques, but all the traffic still passes through a single host. To scale even further, e.g., 10 Gbps, we must avoid the 64-bit/66-MHz PCI bus limitation of 4.2 Gbps. Figure 3 shows how TICKET can be configured with multiple taps to reach even higher network speeds.

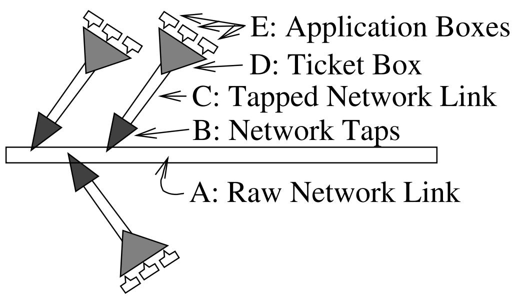

Figure 3. TICKET Configuration with Multiple Taps

This figure shows multiple TICKET boxes attached to the same wire. This removes the requirement that all traffic move through a single host. However, this approach only works if the following conditions hold.

1. Each network tap can be configured to split off only a given subset of traffic, or each NIC is capable of processing link-level packet headers at full link speed.
2. The union of the sets of traffic that the TICKET machines handle covers the original set of traffic.
3. There exists a globally synchronized clock.

The first condition is true when multiple wavelengths are used on a single optical fiber. Such a scenario is becoming more common as the cost of dense wavelength division multiplexing (DWDM) falls. In this case, each tap splits a single wavelength or set of wavelengths from the fiber for processing. (Alternatively, we can connect the network tap to a switch's uplink input and then rely on the switch hardware to demultiplex the input to multiple TICKET machines, as shown in Figure 4.)

The second condition generally follows if the network tap has been properly installed. Subsets of traffic collected by each TICKET machine need not be disjoint (although this is desirable); post-processing of the collected traffic can remove duplicates based on the collected headers and accurate timestamps.

The third condition can be addressed by the network time protocol (NTP) [7]. However, NTP only provides clock accuracy on the order of a millisecond. If data sets are not entirely disjoint (some packets with their associated timestamps are seen in multiple data sets), or if we dedicate a hardware interface solely to synchronization and management, then we can provide the highly accurate synchronization desired.

2.3. Alternate Configuration Two

To this point, we have focused on using commodity hardware to perform network monitoring tasks. Unfortunately, this may not always be possible as the speed of core network links may *always* outstrip commodity hardware. In this case, we can use the alternative TICKET configuration presented in Figure 4.

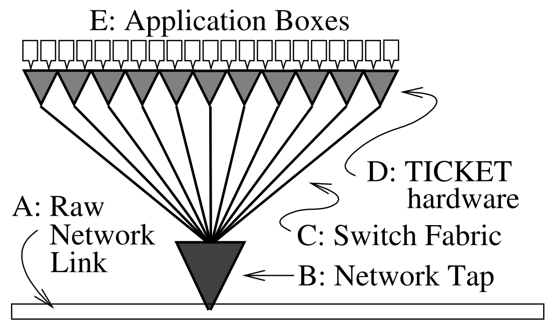

Figure 4. TICKET Switch-Fabric Configuration

Figure 4 shows how TICKET might be implemented in hardware analogous to a network switch (or even using a commodity switch with predefined routes to appropriately split traffic). Instead of a switch uplink, we have a network tap (B) that copies traffic off of a network link (A). Instead of a true many-to-many switch fabric, we require only a simplified one-to-many demultiplexer (C). This fabric would connect to lower-speed network links (just as a switch would) to which we can connect TICKET machines as described earlier during the discussion of port mirroring. Alternatively, simple chips could implement TICKET functionality and be integrated with this device, leaving only the application boxes (E) separate.

This architecture is obviously quite different than the tcpdump/libpcap style of collecting traffic, but it is somewhat similar to the RMON specification [10]. It differs in two fundamental ways: First, TICKET pushes data to the application boxes whereas application boxes must poll the RMON device for data, i.e., pulls the data from the RMON in the RMON specification. Second, the individual network probe of the RMON is replaced with our explicitly parallel TICKET methodology.

3. Implementation

At the moment, the first configuration discussed above suffices for gigabit Ethernet. We implemented this on Intel-based x86 hardware and made heavy use of the source code for the Linux kernel [9]. The software portion of TICKET amounts to a heavily stripped Linux 2.4 kernel with the `init()` function replaced by code that implements the TICKET core functions. This approach leverages a pre-existing code base (scheduler, virtual memory, device drivers, hardware probing) and frees us to focus on writing the most efficient TICKET code possible.

3.1. Creating a Minimal Kernel

We used what initially appears to be a roundabout process to create a stripped Linux kernel, but it provides a much smaller and cleaner end-product. Our initial goal sought to create a source tree containing a truly minimal set of files required to build a kernel for our hardware.

First, we configured and built a monolithic Linux 2.4 series kernel from the stock sources using regular Linux kernel configuration tools, making sure to save the build output for later. This kernel includes support for the hardware we wish to run on, and the build output tells us exactly which source files are used to create our kernel. At this point, we can create a parallel TICKET development tree using just the files actually built into this kernel. This is only for aesthetic and size reasons – a parallel tree is "cleaner" and requires about 15% the space of a stock tree.

This Linux kernel is still a fully functional operating system at this point, stripped only of unused code for a given machine. The next step is to further remove services and functionality unused by TICKET that would reduce performance. For the uniprocessor hardware on which we run TICKET, the only required changes were replacing the `init()` function with a call to our TICKET code[1] removing calls to the `/proc` filesystem, and removing hard-coded hardware probes for non-existent devices. Other implementations may wish to turn off virtual memory handling (which under some versions of Linux will aggressively place pages into disk swap even if there is main memory available, seriously decreasing performance), remove `printk` statements, and so forth.

Now we are left with an absolutely minimal kernel to manage low-level hardware tasks and provide a useful Application Programming Interface (API), which we use to actually program TICKET.

[1] `init()`, causes the first switch to user level, so removing it avoids this switch and thus any user level code execution.

3.2. TICKET Core Details

The core sequence of events for any implementation of TICKET is shown in this list, with further discussion of each point following below:

1. The kernel initializes, probes hardware, and calls TICKET as the sole thread of execution.
2. TICKET parses the kernel command line for options.
3. TICKET brings up network interfaces and waits for link negotiation.
4. TICKET calls a "mode" function; in this case, traffic collection.
5. TICKET executes the following operations "forever"
   (a) Receive a packet and timestamp it.
   (b) Collect information about that packet.
   (c) If output buffer is full, select interface to send and enqueue.
   (d) Send pending output.

First the initial boot-up process occurs, with all the low-level BIOS calls and so forth that are performed by our stripped kernel, before any TICKET code is called.

Next, we make use of the ability to pass kernel command-line arguments. This flexibility allows us to pass TICKET configuration information, such as IP addresses or modes. The "mode" function of Step 4 was added so that the TICKET framework can also be used to perform related tasks, like network flooding for testing, or to enable "Application Layer" functionality that is logically separate from TICKET.

"Forever" in the following step generally means until `ctrl-alt-del` is pressed, or a specially formatted link-level "reboot" packet is sent to the machine. As TICKET runs on general-purpose hardware, one could conceive of security risks by placing such a machine on a network backbone. The passive nature of TICKET and the inability to modify parameters without a reboot currently addresses this problem.

We currently perform timestamps as packets enter the network stack on the *host* CPU; this implicitly assumes that there is a fairly constant delay between packet arrival "off the wire" and the time it enters the network stack via a DMA by the NIC and a host interrupt. Some hardware specific configuration is required to ensure this works, which we discuss in Section 4.

The information collected by the current implementation is as follows:

- 64-bit timestamp
- Length "off-the-wire"
- Ethernet Information:
  - Addresses of Source and Destination
  - Type of service or 802.3 Length
- IP information
  - Addresses of Source and Destination
  - Lengths of Packet and Header
  - Protocol Number
- TCP/UDP information
  - Ports of Source and Destination
  - Length of UDP packet or TCP header

This information is in our opinion the minimal but most useful subset of data available. Some redundancy is included with lengths for checking validity of packets. In practice, this resolves to about 42 bytes of data per packet, which means we are collecting about $6\frac{1}{2}\%$ of the total traffic given a 650-byte average packet size. The information collected can, of course, be easily changed.

At this point, we have reduced our input traffic by a factor of 20, on average. The remaining data is to be sent to the user level machine for saving or further analysis. We wish to minimize the overhead of sending packets on the outgoing link, so we buffer our results until we fill a packet. This packet is then enqueued for sending, and the device driver sends it asynchronously. Given our 42-byte data collection per input packet and 1500-byte Ethernet MTU, we send the results from about 34 input packets per output packet.

4. Performance Evaluation

The only really useful measure of performance is whether the monitor can "keep up" with the network – in our case, that means 1Gbps Ethernet. This section will first discuss what we are testing, then how we are testing, and then the results and analysis for each of the various tests.

4.1. Experimental Dimensions

In our tests, we we consider four representative dimensions:

- TICKET vs. tcpdump
- Slow vs. fast collection machine
- Saving data vs. not saving data
- One interface vs. two interfaces

The first parameter, "TICKET vs. tcpdump," is used to compare the performance of a very commonly used tool with TICKET.

The second parameter, "slow machine vs. fast machine," is used to compare performance of tcpdump or the TICKET kernel on different hardware. The "slow machine" is a dual-processor 400MHz Pentium II machine with a single 32bit/33MHz PCI bus and 128MB of PC-100 memory. The "fast machine" is a uni-processor 933MHz Pentium III machine with both a 64bit/66MHz PCI bus and a 64bit/33MHz PCI bus, and 512MB of PC-133 memory. Both machines have IDE drives, one or more AceNIC Gigabit Ethernet cards with the Tigon II chipset, and a single EtherExpress Pro 100Mb Ethernet card for management.

The third parameter, "saving data vs. not saving data", is only for tcpdump. It means whether data is saved to a real file via `tcpdump -w FILE` or whether it is thrown away via `tcpdump -w /dev/null`. TICKET always saves data, but not on the host collecting data; testing tcpdump without saving data mimics this behavior.

The case of tcpdump not saving data is analogous to a user level implementation of TICKET; it does not save data locally and we can think of it using another machine to save that data instead. This analogy allows us to quantify the effect of each of the major differences between TICKET and tcpdump: (a) single-machine vs. shared among machines, and (b) user mode vs. kernel mode execution.

Comparing tcpdump saving and tcpdump not saving can be thought of as a comparison between the user level tcpdump and a user level implementation of TICKET – showing the effect of single-machine vs. multiple machine execution.

Comparing tcpdump not saving data and TICKET can be thought of as a comparison of user level and kernel level TICKET implementations – showing the effect of user mode versus kernel mode execution.

The fourth parameter, "one interface vs. two interfaces," allows us to eliminate possible bottlenecks with a single interface. Most switches allow configuration of a load-sharing group, where multiple interfaces are grouped into a logical interface, and data is striped among them. This idea will show how an approach scales to utilize multiple input and output interfaces.

4.2. Experimental Setup

We build a network in the simplest TICKET configuration (a single kernel machine connected to a network tap with one or more user level machines post-processing data) and measure how much traffic is analyzed while varying each of the parameters listed above. The more complex configurations listed in Section 2 are not yet required for today's networks.

The dump machine is only used by TICKET; tcpdump runs entirely on the collection machine discussed in the previous section. The hardware for the dump machine is an Intel L440GX+ motherboard with dual 500MHz Pentium III processors and 1GB of 100MHz ECC DRAM. It contains 1.2TB of disk space that is split among three RAID-5 arrays of size 375MB, 375MB, and 525MB. Each RAID is made up of several 80GB Maxtor DiamondMax hard drives and a 3Ware Escalade 6800 8-channel Ultra ATA RAID controller. The largest RAID-5 array, one of the 375MB arrays, and AceNIC Gigabit Ethernet card share a 32bit/33MHz PCI bus (due to space constraints in the case), while the remaining 375MB array is on a separate 64bit/33MHz PCI bus. Sharing the smaller bus among so many cards creates a bottleneck; thus, we use a full gigabyte of memory to allow a large amount of buffering.

The software we are using is TICKET, version 1.0 alpha[2] When comparing these values with tcpdump/libpcap, we use the prepackaged Debian Linux stable versions – libpcap 0.6.2-2 and tcpdump 3.6.2-2.

The physical configuration of our network is shown in Figure 5. We set up two TICKET machines in "traffic generation" mode; using the kernel level framework to generate almost exactly 500Mbps of traffic in 1066-byte UDP packets (1024 bytes data, 42 bytes of UDP, IP, and Ethernet headers). This traffic aggregates to 1Gbps at our GigE switch (Extreme Networks' Summit 7i), which then forwards it to the collection box running TICKET or tcpdump. The collection box processes the data and either sends it to the dump RAID box via a crossover cable to be saved to disk or saves it locally. This data can then be checked for errors or packet loss, or any other arbitrarily complex analysis. All NICs in these systems are of the copper variety.

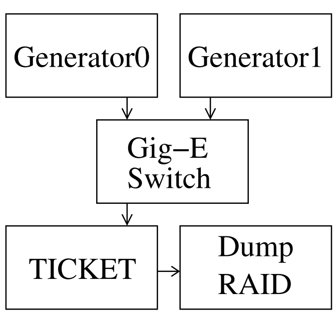

Figure 5. TICKET Verification Test Setup

For simplicity, this figure only shows gigabit Ethernet links. In practice, each machine is also part of a separate fast Ethernet network for management, and some have serial connections for remote consoles.

[2]This will be released to the public under the Gnu Public License (GPL) in the near future. Check our web site, http://www.lanl.gov/radiant.

4.3. Experimental Results

We present pairs of graphs showing the same test performed on the "slow" and "fast" machines. Measurements are made by observing statistics at the switch and at the source "flooder" machines and by the output of TICKET or tcpdump.

Figure 6 compares the amount of data received by the slow machine and that observed by tcpdump. Results using a single, unshared interface and two, load-sharing interfaces are presented.

The first noticeable feature is that the amount of data observed does not change significantly when two interfaces are used. In this case, the disk is the bottleneck, and a single interface is sufficient to saturate disk bandwidth. Two interfaces with an instance of tcpdump observing each makes no difference because there is only a single disk. The addition of a second disk would likely increase the amount of data saved.

The second noticeable feature is that the amount of data received does not double when a second interface is used; the CPU and accessory bandwidth become the bottleneck in this case. Thus, while the machine now receives about 65% of the traffic sent to it, it still observes only about 15%.

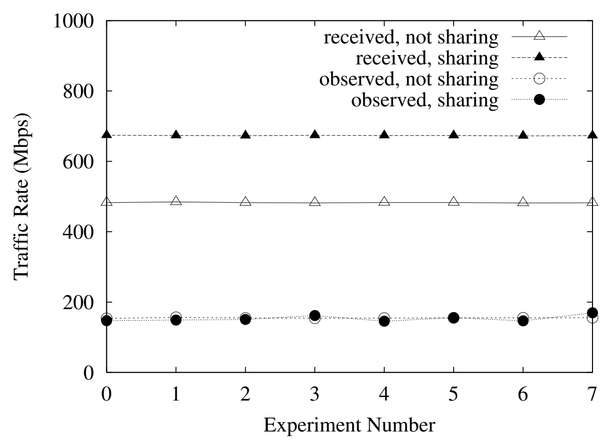

Figure 6. Tcpdump, Slow Machine, Saving Data

Figure 7 presents the same experiment on the fast machine. This figure shows approximately the same observation results as for the slow machine but with a 20-25% improvement in traffic observed with or without load-sharing. The interesting feature is that the amount of data received by the machine is nowhere near that received by the "slower" machine. This is due primarily to the fact that the "slower" machine has two CPUs; one can handle network interrupts to receive data while the other runs tcpdump to observe data. The "faster" machine has a single processor that must do both, the amount of data observed is about the same as the amount of data received.

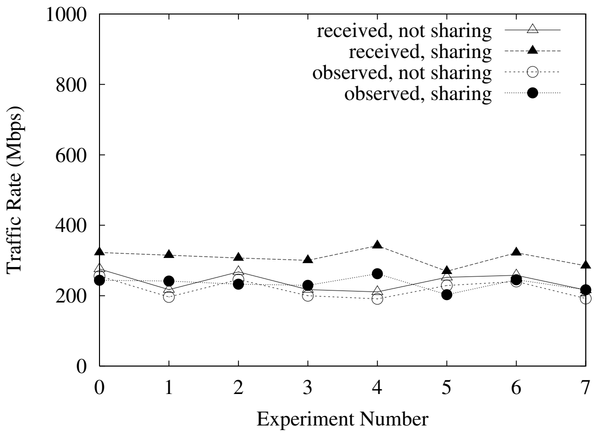

Figure 7. Tcpdump, Fast Machine, Saving Data

Figure 8 continues with the results for tcpdump, this time throwing away data by writing it to /dev/null. "Observed" in this test is somewhat of a misnomer, as observation is simply counting each packet as it is thrown away.

Again, the amount of data observed does not change significantly when two interfaces are used. This time it is not the disk that is the bottleneck but the massive number of user level to kernel level copies and context switches (several per packet). Even throwing away all data, we still only count about 50% the packets that have been sent. Again, load-sharing helps increase the number of packets received, but only to 80% of what was sent.

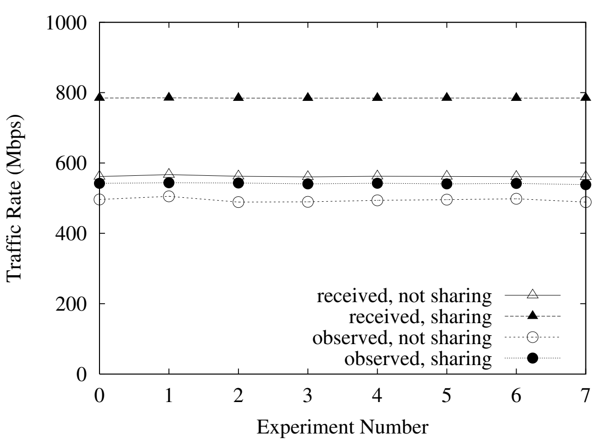

Figure 8. Tcpdump, Slow Machine, Not Saving Data

Figure 9 shows the same experiment on the faster machine. As in Figure 7, we see that the amount of data observed and received is about the same for each test. For the single NIC test, we receive and count about 65% of the data sent. For the dual NIC test, we receive all the traffic sent, but only observe about 95% of it. This figure shows that when we remove the disk bottleneck, the faster machine's one processor is able to significantly outperform the slower machine's two processors.

Comparison of Figure 8 with Figure 6 and Figure 9 with Figure 7 shows the benefit that would be gained by a user level TICKET striping over multiple machines. Figure 9 indicates that a user mode TICKET with load-sharing functionality would almost be able to keep up at gigabit speeds on the faster hardware.

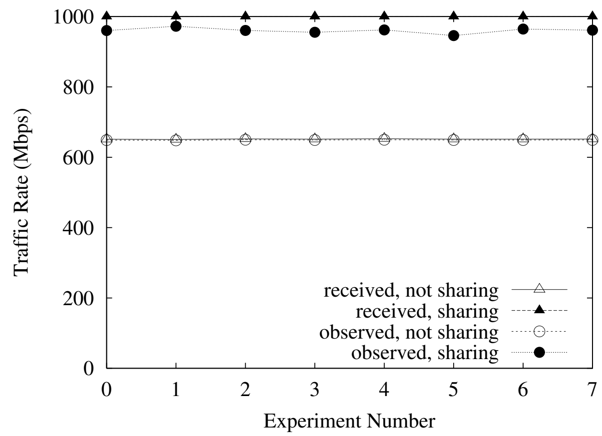

Figure 9. Tcpdump, Fast Machine, Not Saving Data

Figure 10 presents the first TICKET data, as compared to tcpdump. The tcpdump data is the same as in Figure 6. From now on, we only present the amount of data observed. The amount of data received by tcpdump has already been shown, and the amount received by TICKET is within 0.5% of the amount of data observed in all cases.

We see that without load-sharing, TICKET observes about 55% of traffic sent, nearly three and a half times the amount tcpdump is able to observe. With load-sharing, TICKET observes over 70% of traffic sent, over four and a half times the amount tcpdump observes.

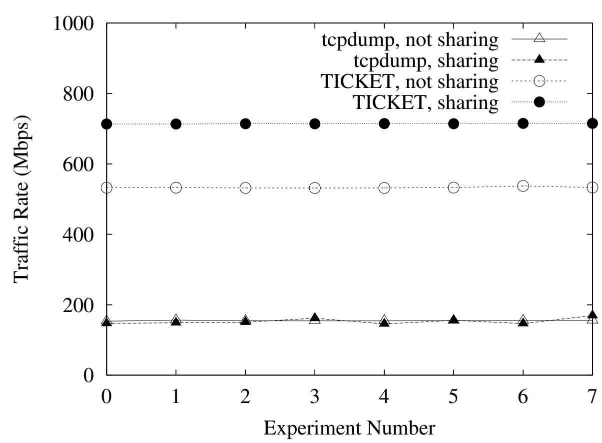

Figure 10. Tcpdump and TICKET, Slow Machine, Saving Data

Figure 11 shows the same experiment with the faster machine. Without load-sharing, TICKET observes almost 70% of the data sent. This is about three times the amount that tcpdump can observe. With load-sharing, TICKET observes 100% of the data sent! This graph also shows the best case for tcpdump, observing only about 25% of data sent – and thus TICKET outperforms tcpdump by a factor of four.

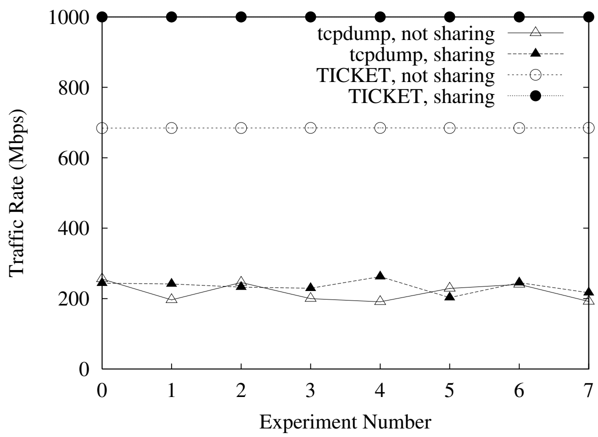

Figure 11. Tcpdump and TICKET, Fast Machine, Saving Data

Figure 12 contains data from in Figures 8 and 10, while Figure 13 contains data from Figures 9 and 11. These new graphs compare the cost of user level versus kernel level execution.

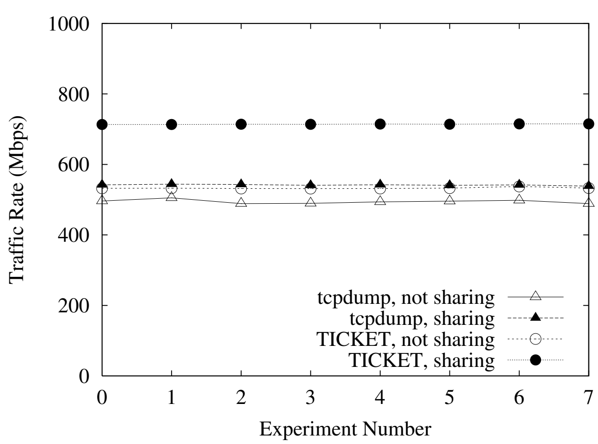

Figure 12. Tcpdump and TICKET, Slow Machine, Not Saving Data

For a single NIC, performance is similar between a kernel level TICKET and a user level TICKET (tcpdump without saving). This is because the bottleneck is primarily the NIC and its inability to keep up at gigabit speeds. When load-sharing is introduced, that bottleneck is removed. Then the limiting aspects of user level code become more prominent and the kernel mode TICKET significantly outperforms the user level tcpdump.

Figure 13 shows the same data for the faster machine. Performance is similar between a kernel level TICKET and tcpdump without saving for both the single NIC and load-sharing cases. The absence of a difference shows that given a faster processor, the user level to kernel level copies and context switches are less significant.

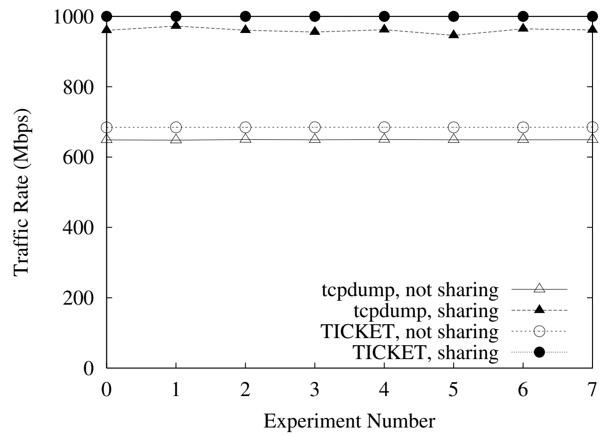

Figure 13. Tcpdump and TICKET, Fast Machine, Not Saving Data

5. Conclusion

We have provided evidence that TICKET will consistently outperform one commonly used tool by an order of magnitude. Less commonly used custom hardware approaches may offer similar functionality, but they cost up to $200,000 whereas our system runs on commodity hardware that costs less than $2,000. We have also used the system in real-world situations and found that the experimental results presented here are a good predictor of actual performance.

In sum, we have discussed a method for collecting traffic that offers many benefits: it is scalable, reliable, accurate, highly precise in its measurements, and enables many types of applications that were previously impossible. We have provided further motivation for this approach by an in-depth consideration of computing trends, other work, and experimental evidence.

References

[1] CAIDA. CoralReef Software Suite. http://www.caida.org/tools/measurement/coralreef.

[2] K. Coffman and A. M. Odlyzko. Internet Growth: Is there a "Moore's Law" for data traffic? http://www.research.att.com/ amo/doc/internet.moore.pdf.

[3] G. Combs and The Free Software Community. The Ethereal Network Analyzer, July 1998. http://www.ethereal.com/.

[4] Lawrence Berkeley National Laboratory Network Research. TCPDump: the Protocol Packet Capture and Dumper Program. http://www.tcpdump.org/.

[5] R. Minnich, D. Burns, and F. Hady. The memory-integrated network interface. *IEEE Micro*, February 1995.

[6] G. E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 38(2), April 1965. http://www.intel.com/research/silicon/moorespaper.pdf.

[7] NTP. Network Time Protocol (NTP). http://www.eecis.udel.edu/ ntp.

[8] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of ACM SIGMOD Conference on the Management of Data*, pages 109–116, 1988.

[9] L. Torvalds and The Free Software Community. The Linux Kernel, September 1991. http://www.kernel.org/.

[10] S. Waldbusser. Remote Network Monitoring Management Information Base (RFC1757), February 1995.