Passive and Active Measurement Workshop (PAM2002),
Fort Collins, Colorado, March 2002. LA-UR 02-0808

Monitoring Protocol Traffic with a MAGNeT

Mark K. Gardner[†], Wu-chun Feng[†‡], Jeffrey R. Hay[†]
{mkg, feng, jrhay}@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*— Network researchers have traditionally focused on monitoring, measuring, and characterizing traffic in the network to gain insight into building critical network components (e.g., protocol stacks, routers, switches, and network interface cards). Recent research suggests that additional insight can be obtained by monitoring traffic at the application level (i.e., before traffic is modulated by the protocol stack) rather than in the network (i.e., after traffic is modulated by the protocol stack). Thus, we present MAGNeT: Monitor for Application-Generated Network Traffic, a toolkit that captures traffic generated by the application (as it traverses the protocol stack) rather than traffic in the network.**

MAGNeT provides the capability to monitor protocol-stack behavior, construct a library of traces of application-generated traffic, verify the correct operation of protocol enhancements, troubleshoot and tune protocol implementations, and perhaps even replace `tcpdump`.

In addition, we have extended MAGNeT to instrument CPU context switches as an example of how the general kernel monitoring mechanisms of MAGNeT can be used to monitor *any* kernel event in the operating system.

***Index Terms*— monitor, measurement, network protocol, traffic characterization, TCP, `tcpdump`, MAGNeT, traces, application-generated traffic, CPU scheduler.**

I. INTRODUCTION

Network researchers often use traffic libraries such as `tcplib` [1], network traffic traces such as those at [2,3], or network models such as those found in [4] to obtain insight into network-protocol operation and to focus their network experiments. However, such libraries, traces, and models are based on measurements made by `tcpdump` [5] (or similar tools like PingER [6], NLANR Network Analysis Infrastructure [7], NIMI [8] or CoralReef [9]), meaning that the traffic an application sends on the network is captured only *after* having passed through TCP (or more generally, any protocol stack) and into the network. That is, the tools capture traffic *on the wire* (or *in the network*) rather than at the application level. Thus, the above tools cannot provide any protocol-independent insight into the actual traffic patterns of an application.

Researchers have traditionally designed and tested network protocols using either (1) "infinite" file transfers or (2) traffic traces which have already been modulated by a protocol stack. The first is appropriate if bulk data transfers constitute the majority of the traffic. But networks are no longer primarily filled with file transfer protocol (FTP) traffic. They include substantial amounts of hypertext transfer protocol (HTTP) and streaming multimedia traffic. The second is acceptable if the differences between application-generated traces and network-captured traces are negligible. However, as we will show in this paper, the differences in the traces can be substantial, indicating that the protocol stack adversely modulates the application-generated traffic patterns. Hence tools for obtaining application-generated traffic traces are needed.

To determine the application-generated traffic patterns before being modulated by a protocol stack,

This work was supported by the U.S. Dept. of Energy's Laboratory-Directed Research & Development Program and the Los Alamos Computer Science Institute through Los Alamos National Laboratory contract W-7405-ENG-36. Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DOE, Los Alamos National Laboratory, or the Los Alamos Computer Science Institute.

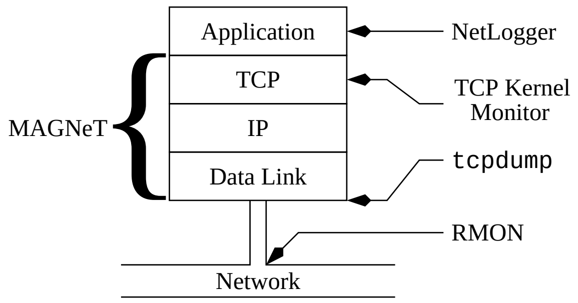

Fig. 1. Monitoring Points of Various Tools

as well as to determine the modulation caused by each layer of the stack, we present the Monitor for Application-Generated Network Traffic (MAGNeT). MAGNeT differs from existing tools in that traffic is monitored not only upon entering and leaving the network but also throughout the entire network-protocol stack.

MAGNeT is not limited to monitoring network protocol stacks. It provides a general framework for monitoring any kernel event. As we will show, it can be used to monitor even time-sensitive events such as context switches.

A. Related Work

As Figure 1 shows, MAGNeT differs from `tcpdump`-like and RMON tools in that it makes fine-grained measurements throughout the entire protocol stack, not just at the network wire level. While the TCP kernel monitor [10] is similar to MAGNeT, MAGNeT differs in at least two ways. First, MAGNeT can be used anywhere in the protocol stack, not just for monitoring TCP. Second, MAGNeT monitors a superset of the data that the TCP kernel monitor does.

NetLogger [11] collects, correlates, and presents information about the state of a distributed system. It includes tools for instrumenting applications, host systems, and networks. It also presents tools for visualizing the collected data. However, it requires recompilation or relinking of applications. Because of its focus on overall system dynamics, NetLogger is better than MAGNeT at presenting an overall view of complex distributed system behaviors that are the result of the interaction of multiple components such as network, disk and CPU activity.

In contrast, MAGNeT monitors all applications without modification. It does not require applications to be recompiled or relinked. MAGNeT also provides greater detail about the state of the network protocol stack (or operating system) than NetLogger. Furthermore, MAGNeT's timestamps are several decimal orders of magnitude more accurate. Thus, we view MAGNeT as complementary to NetLogger and plan to make MAGNeT's output compatible to leverage NetLogger's visualization tools.

II. MAGNeT Design

The design of MAGNeT focuses on two primary goals: accurate timestamps and transparency to the end user when run in a production environment. We achieve accurate time measurement by using the CPU cycle counter (available in modern microprocessors) to record timestamps with cycle-level granularity. To provide transparency to the end user, we implement the core MAGNeT functionality as an operating system (OS) patch. This patch creates a circular buffer in kernel memory and places function calls throughout the networking stack to record appropriate information as data traverses the stack. A user-space program, packaged with the MAGNeT toolkit, periodically empties this kernel buffer, saving the binary data to disk. For the post-processing of data, a set of data-analysis tools translates the binary data into human-readable form. Finally, to complete the MAGNeT toolkit, a library of scripts automates the collection of data from a set of MAGNeT-ized hosts.

A. MAGNeT in Kernel Space

Figure 2 provides a high-level overview of the operation and data flow of MAGNeT. Applications make `send()` and `recv()` system calls during their course of execution to send and receive network traffic. These calls eventually make use of TCP, IP, or other network protocols in the kernel to transfer data on to the network. Each time a protocol event occurs, the MAGNeT-ized kernel makes a call to the MAGNeT recording procedure (which in our implementation is called `magnet-add()`). This procedure saves relevant data to a circular buffer in kernel space, which is then saved to disk by the MAGNeT user-level application program called `magnet-read`.

The MAGNeT kernel patch adds several functions to Linux 2.4. The function `magnet_add()` adds a data point to a circular buffer that is pinned in kernel

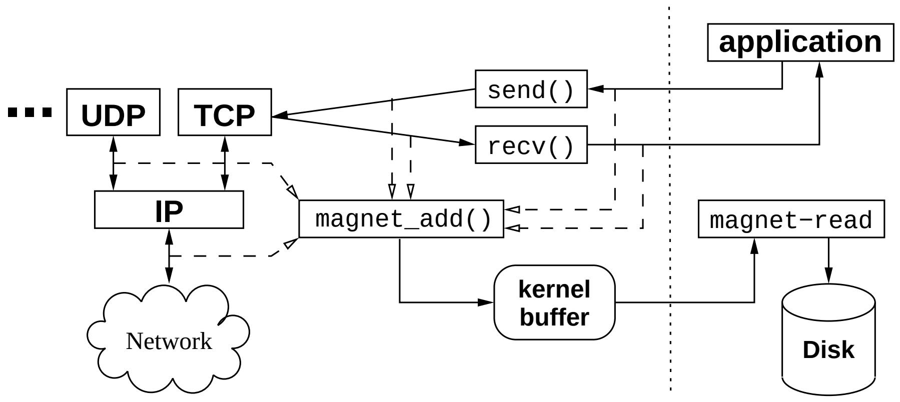

Fig. 2. Overview of MAGNeT Operation

memory. This function is optimized so that each instrumentation call uses as few resources as possible and can be called anywhere in the protocol stack. In addition, a new file is added to the `/proc` file system at `/proc/net/magnet`. This file may be read by any user to determine the current state and parameters of the MAGNeT kernel code.

Figure 3 shows the data structure that is appended to the kernel buffer as a MAGNeT instrumentation record at each instrumentation point. The `sockid` field contains a unique identifier for each connection stream, providing a way to separate individual data streams from a trace while protecting the privacy of the application and user. The `timestamp` field holds the value read from the CPU cycle counter and also synchronizes MAGNeT's kernel- and user-space processes. The `event` field keeps track of the type of event a particular record refers to, e.g., `MAGNET_IP_SEND`. The `size` field contains the number of bytes that were transferred during the event. Finally, the `data` field (an optional field selected at kernel-compile time) is a union of various structures in which information specific to particular protocols can be stored. This field provides a mechanism for MAGNeT to record protocol-state information along with event transitions. Figure 14, in the appendix, shows the union members for TCP and IP events. (For more details on MAGNeT's implementation, see [12]).

B. MAGNeT in User Space

The user-level interface to MAGNeT consists of three application programs (`magnet-read`, `magnet-parse`, and `mkmagnet`), a special device file to facilitate kernel/user communication, and automating scripts. `magnet-read` saves data from

```
struct magnet_data {
  void *sockid;
  unsigned long long timestamp;
  unsigned int event;
  int size;
  union magnet_ext_data data;
}; /* struct magnet data */
```

Fig. 3. The MAGNeT Instrumentation Record

the kernel's buffer, which is exported via the special device file, to a disk file, and `magnet-parse` translates the saved data into a human-readable form. `mkmagnet` is a small utility program to create the files that `magnet-read` requires to operate. The scripts included with the MAGNeT distribution allow the operation of MAGNeT to be fully automated and transparent to the end user.

C. Kernel/User Synchronization

The MAGNeT kernel patch exports a circular buffer to user space via shared memory. Since the kernel and user processes access the same area of physical memory, MAGNeT coordinates accesses by the two processes using a field of the instrumentation record as a synchronization flag between the MAGNeT user and kernel processes, as shown in Figures 4 and 5. Specifically, our initial implementation of MAGNeT uses the `timestamp` as the synchronization field.

Before writing to a slot in the buffer, the MAGNeT kernel code checks the synchronization field for that slot. If the field indicates that the slot has not yet been copied to user space (e.g., `timestamp` field is non-zero), the kernel buffer is full. In this case, the kernel code increments a count of the number of

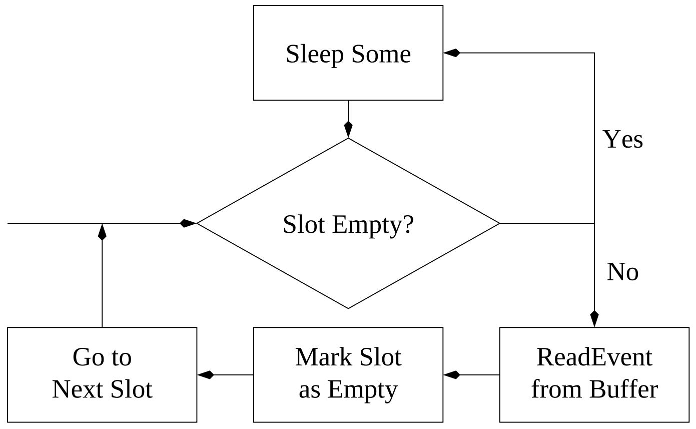

Fig. 4. MAGNeT User Operation

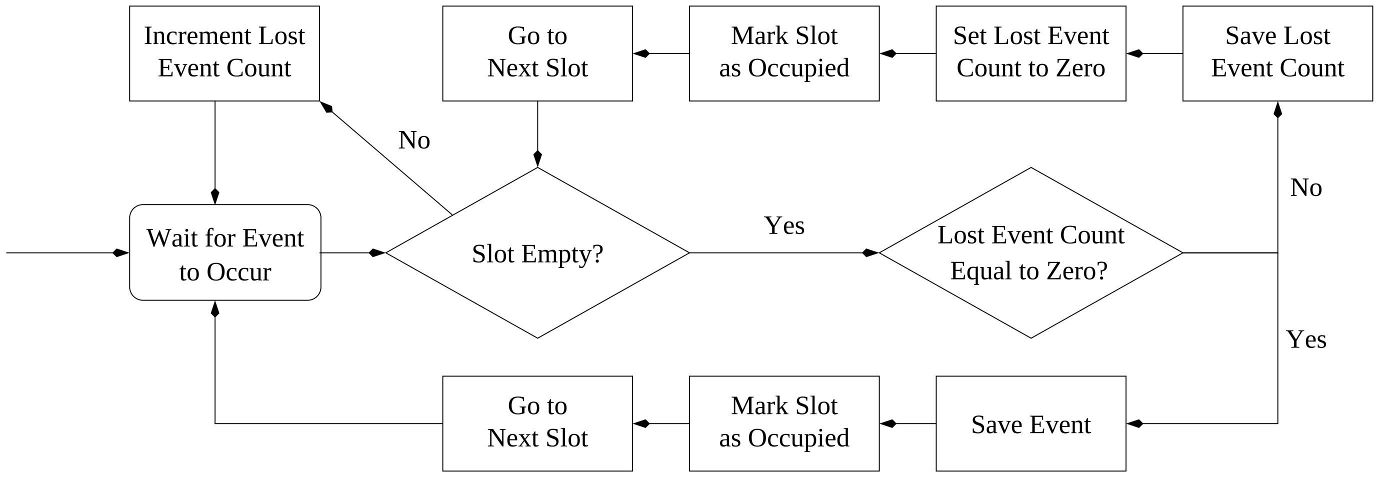

Fig. 5. MAGNeT Kernel Operation

instrumentation records that could not be saved due to the buffer being full. Otherwise, the kernel code writes a new instrumentation record and advances to the next slot in the buffer.

The user application accesses the same circular buffer via kernel/user shared memory. When the synchronization field at the current slot indicates the slot contains a valid record, the application reads the entire record and resets the synchronization field (e.g., resets the `timestamp` field to zero) to signal to the kernel that the slot is once again available. The application then advances to the next slot in the buffer.

When the kernel has a non-zero count of unsaved events and buffer space becomes available, the kernel writes a special record to report the number of instrumentation records that were not recorded. Thus, during post-processing of the data, the fact that events were lost is detected at the appropriate place within the data stream. Our experience to date indicates that while unrecorded events are possible, they rarely occur during the monitoring of actual users. We will return to the subject of lost events in Section III-A.2.

D. *MAGNeT Timestamps*

To ensure the greatest accuracy possible, MAGNeT uses the cycle counter available on contemporary microprocessors as the source of its timestamps. MAGNeT obtains this information via the kernel's `getcyclecounter()` function, which keeps the MAGNeT code hardware-independent. Given the speed of the processor, the difference between two cycle counts can be converted to an elapsed time. Given a time referent, elapsed time can be converted to a date and time.

MAGNeT exports the processor speed and native bit-order of the trace in the first record of the trace so that `magnet-read` or other user-space tools can convert the timestamps to elapsed time. MAGNeT also exports the starting date and time (as seconds since the Unix epoch) so that elapsed time can be converted to a date and time.

	Configuration
1	Linux 2.4.3
2	Linux 2.4.3 w/MAGNeT
3	Linux 2.4.3 w/MAGNeT, `magnet-read` on receiver
4	Linux 2.4.3 w/MAGNeT, `magnet-read` on sender
5	Linux 2.4.3, `tcpdump` on receiver
6	Linux 2.4.3, `tcpdump` on sender

TABLE I

TEST CONFIGURATIONS

	100 Mbps		1000 Mbps	
	Throughput (Kbps)		Throughput (Kbps)	
1	94.14	± 0.00	459.48	± 1.63
2	94.13	± 0.01	452.46	± 1.82
3	90.79	± 0.82	444.31	± 1.66
4	90.69	± 0.88	440.24	± 2.11
5	89.39	± 1.48	290.68	± 15.64
6	89.04	± 0.84	343.22	± 18.71

TABLE II

PERFORMANCE OF MAGNeT VS. `tcpdump`

III. MAGNeT PERFORMANCE ANALYSIS

To determine the overhead of running MAGNeT, we measure the maximum data rate and the CPU utilization between a sender and receiver with and without MAGNeT. For comparison, we also measure the overhead of running `tcpdump`. In total, the six configurations shown in Table I are compared.

Our baseline configuration runs between two machines with stock Linux 2.4.3 kernels. The second configuration uses the same machines but with the MAGNeT patches installed. Although present in memory, MAGNeT records are not saved to disk. The third configuration is the same as the second except `magnet-read` runs on the receiver to drain the MAGNeT buffer. The fourth configuration is also the same as the second, but with `magnet-read` on the sender. For the fifth and sixth configurations, `tcpdump` is run on stock Linux 2.4.3 kernels. The fifth configuration runs `tcpdump` on the receiver, while the sixth runs `tcpdump` on the sender. All configurations are tested on both 100 Mbps and 1000 Mbps Ethernet networks.

We conduct the tests between two identical dual 400 MHz Pentium IIs with NetGear 100 Mbps and Alteon 1000 Mbps Ethernet cards. MAGNeT is configured to record only the transitions between protocol stack layers, not the optional information about the packets and the protocol state. The default 256 KB kernel buffer is also used to store event records.

For a workload, we use `netperf` [13] on the sender to saturate the network.[1] We minimize the amount of interference in our measurements by eliminating all other network traffic and minimizing the number of processes running on the test machines to `netperf` and a few essential services.

[1]The command we use is "`netperf -P 0 -c <local CPU index> -C <remote CPU index> -H <hostname>`"

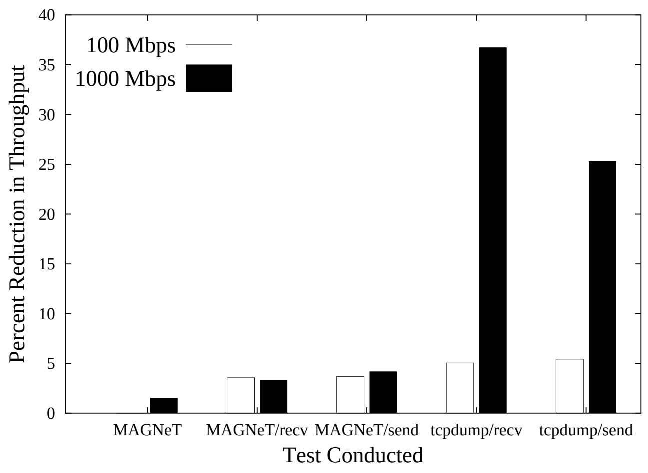

Fig. 6. Percent Reduction in Network Throughput

A. Performance

We compare the performance of MAGNeT with `tcpdump` as the closest commonly-available monitoring tool even though MAGNeT records a different set of information than `tcpdump` (i.e., MAGNeT records application and protocol stack-level traffic while `tcpdump` only records network-wire traffic). By default (and as tested) `tcpdump` stores the first 68 bytes of every packet. MAGNeT, by default, stores a 24-byte record for each protocol of the four layers in the stack. Thus MAGNeT records 96 bytes for every packet.

Table II and Figure 6 present the performance of MAGNeT and `tcpdump` for our tests. Along with the mean, the width of the 95% confidence interval is given.

1) Network Throughput: The kernel-resident portion of MAGNeT executes whether information is being saved to disk or not. Table II shows that "Linux 2.4.3 w/MAGNeT" over 100 Mbps and 1000 Mbps Ethernet incurs negligible overhead when data is not being saved to disk. With `magnet-read` active on the receiver or sender, MAGNeT incurs less than a 5% reduction in network throughput on a *saturated* network. Furthermore, the penalty is nearly constant regardless of network speed. In contrast, while `tcpdump` incurs roughly the same penalty as MAGNeT over 100 Mbps networks, the penalty increases to 25%-35% of total throughput at 1000 Mbps. Thus, MAGNeT scales better than `tcpdump`.

2) Event Loss: Analysis of the MAGNeT-collected data for our tests reveals that MAGNeT occasionally fails to record events at high network utilization. On a *saturated* network, MAGNeT, in its default configuration, was unable to record approximately 3% of the total events for the 100 Mbps trials, while for the 1000 Mbps tests the loss rate approached 15%. These losses are due to the 256 KB buffer in the kernel filling before `magnet-read` is able to drain it.

By comparison, loss rates for `tcpdump` are significantly higher than for MAGNeT. Under the same test conditions, average packet-loss rates for `tcpdump` are around 15% on a saturated 100 Mbps network, i.e., five times higher than MAGNeT in its default configuration.

Because `tcpdump` does no buffering, loss rates will increase as network speeds increase. In contrast, if MAGNeT's loss rate is too high, it can be adjusted to an acceptable level via the mechanisms discussed below; `tcpdump` lacks such adjustability.

As noted in [14], our toolkit provides two methods for reducing the event loss rate: (1) increasing the kernel buffer size or (2) reducing the time `magnet-read` waits before draining the kernel buffer. Figure 7 shows the effect of these parameters on event loss rate for the 100 Mbps saturated network tests.

Increasing the kernel buffer size reduces MAGNeT's event loss rate down to virtually no lost events under any network load with a 1 MB buffer. However, because this buffer is pinned in memory, a large buffer also reduces the amount of physical memory available to the kernel and applications.

Adjusting the amount of time `magnet-read` waits before draining newly accumulated records also affects the performance of MAGNeT. Shorter delays cause the buffer to be drained more frequently, thus reducing the chance of lost events. However, shorter delays create more work (in terms of CPU usage and, possibly, disk write activity), and thus may interfere with the system's normal use.

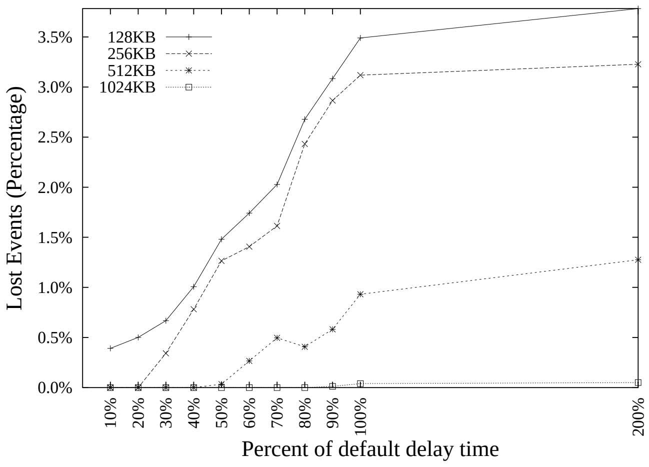

Fig. 7. MAGNeT's Event-Loss Rate, 100 Mbs Ethernet

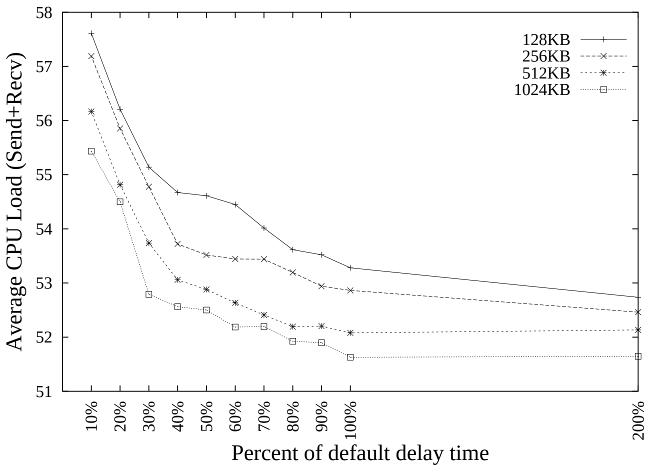

Fig. 8. MAGNeT's Average CPU Utilization, 100 Mbs Ethernet

Figure 8 shows the average CPU utilization for different delays and buffer sizes with MAGNeT running on the sending machine. (The high CPU utilization reported in this graph is due to our test procedure of flooding the network with `netperf`, which places an unusually high load on the sender CPU.) As the results show, CPU utilization is relatively insensitive to the range of kernel buffer sizes tested but it is sensitive to changes in delay.

IV. Applications of MAGNeT

The MAGNeT toolkit provides a transparent method of gathering application traffic data on individual machines. MAGNeT meets its goal of recording application-generated traffic patterns while causing minimal interference. In this section, we provide examples of how MAGNeT-collected information can be utilized.

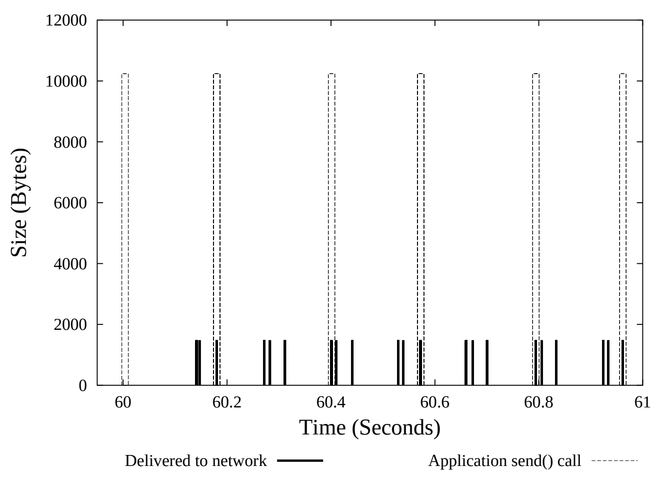

Fig. 9. MAGNeT FTP trace

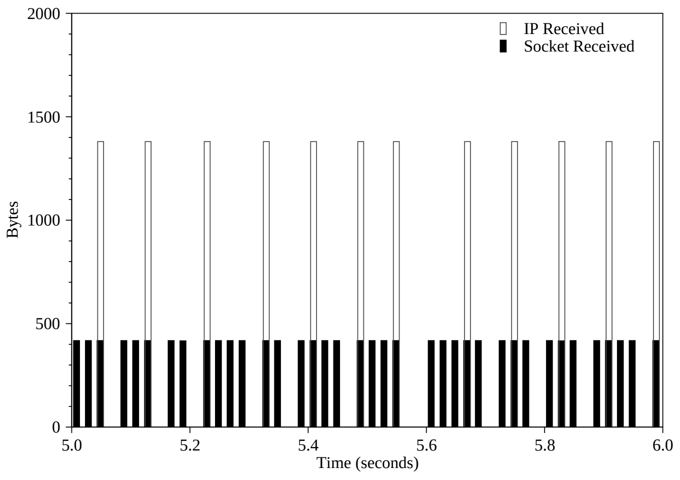

Fig. 10. MAGNeT Trace of MPEG-1 Layer 3 Audio Stream

A. *Traffic Pattern Analysis*

We can use MAGNeT-collected traces to investigate differences between the traffic generated by an application and that same traffic as it appears on the network (i.e., after modulation by a protocol stack). As a simple example, we consider a trace of a FTP session from our facility in Los Alamos, NM to a location in Dallas, TX. Figure 9 shows a one-second MAGNeT trace, taken one minute into the transfer.

As can be seen from the graph, the FTP application attempts to send 10 KB segments of data every 20 milliseconds, but the protocol stack (TCP and IP in this case) modulates the traffic into approximately 1500-byte packets, the maximum payload size on Ethernet networks, at intervals of varying duration. The variable spacing of the traffic intervals is caused by TCP waiting for positive acknowledgements before sending more traffic.

If we send the traffic stream *as it was delivered to the network* through another TCP stack, as would be the case when a firewall breaks a logical connection into two physical connections, we see further modulation. Each subsequent run of network-delivered traffic through TCP further modulates the traffic, as shown in Table III. Thus, we see that TCP significantly perturbs traffic patterns, even when the traffic pattern has previously been shaped by TCP. This result implies that wire-level traffic traces do not represent the true networking requirements of end-user applications.

TABLE III
EFFECT OF MULTIPLE TCP STACKS

Trial	Data Size	Inter-packet Spacing (sec)
Application	3284	0.124
1st TCP stack	1016	0.045
2nd TCP stack	919	0.037
3rd TCP stack	761	0.079
4th TCP stack	723	0.122

The next example shows a dramatically different traffic pattern illustrating that the demands placed upon the network depend upon the application.

In this example, we monitor network traffic on a machine playing a 128 Kbps MPEG-1 Layer 3 (MP3) audio stream from mp3.com. To initiate the transfer, the MP3 player requests the desired stream. The server then begins streaming the data at a uniform rate. Figure 10 shows one second of a MAGNeT trace taken five seconds after the audio stream begins.

The figure graphically shows when the IP layer receives data from the device driver and when the socket layer of the kernel receives data from the TCP layer. The application's network requirements are for small blocks of 418 bytes, corresponding to the frames of the MP3 audio stream, in bursts which are non-uniformly spaced but which average 21 ms apart. The average data rate delivered to the application is 127.1 Kbps. On the other hand, the network delivers 1380 bytes of data regularly every 79 ms for an average data rate of 139.9 Kbps.

This behavior is in contrast to the FTP application shown earlier where the application requests large blocks of data periodically and the network fragments the data into smaller units and transmits them with irregular spacing. Thus we see that application-

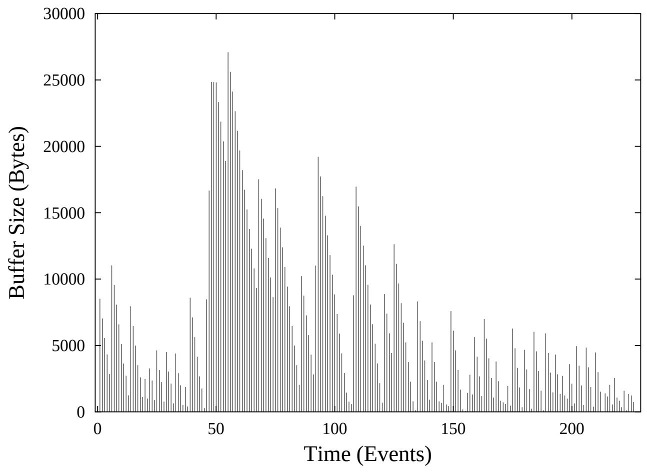

Fig. 11. Size of Sending TCP Buffer

generated requests vary widely, depending upon the application, and do not resemble the traffic on the network.

B. Resource Management

With the optional `data` field compiled in, MAGNeT can return snapshots of the complete protocol state during the execution of real applications on live networks. Previously, this information was only available in simulation environments such as *ns*. This kind of data is invaluable when planning proper resource allocation on large computing systems.

For example, Figure 11 shows the size of the sending TCP buffer during the streaming of an MPEG movie. The buffer contains data recorded as "sent" by the application, but not yet actually delivered to the network by TCP. It reaches a maximum size of around 30 KB but averages 6.5 KB for the life of the connection. With this kind of information, a resource allocation strategy which conforms to the true needs of applications may be developed.

C. Network-Aware Application Development

As discussed in Section II, MAGNeT captures data which network-aware applications can use to appropriately tune their performance. In our implementation, any application is able to open the MAGNeT device file and map the MAGNeT data-collection buffer to a portion of its memory space. Thus, a daemon may be developed which monitors the MAGNeT-collected data and provides a summary of the data for specific connections at the request of network-aware applications. This strategy consolidates all network-monitoring activity to amortize the overhead across all network-aware applications running on the system.

D. Kernel Monitoring

Besides being useful for monitoring the network protocol stack, MAGNeT can also be used to monitor other kernel events. As an example, we extend MAGNeT to record the scheduling behavior of the operating system (OS). Clearly, such information is useful to OS designers and implementors.

Here, we focus on the following events: process creation (`fork`), scheduling (`schedule`) and termination (`exit`). From the MAGNeT traces, we calculate the duration of the context switches. By context-switch duration we mean the time it takes for the kernel to decide which process to execute next and to reload the process' state.[2] Modifying MAGNeT for this purpose requires a total of 26 lines of code to be added to four files.

For the experiment, we run one of the machines in Section III with the MAGNeT-ized kernel containing the monitoring extensions to the scheduler. The baseline workload has only the usual system processes (the "idle" test). The next workload adds a process spinning in an infinite loop doing nothing (CPU-bound test).

Figure 12 shows the duration of every context switch that the system executes as a function of the time since the beginning of the "idle" system test. The average context switch time for CPU #0 is around 2 μs corresponding to the fast path through the scheduler with occasional excursions to approximately 4 μs due to taking the slow path instead. On the other hand, context switches for CPU #1 are approximately 4-6 μs likely due to contention on the spin lock protecting the ready process queue.

Figure 13 shows the same graph but with the CPU-bound process running. The average context switch time for CPU #0 now oscillates between 2 μs and 6 μs with a period of around 2 sec. Likewise, the average context switch time for CPU #1 also oscillates but 180 degrees out of phase. This curious behavior is due to a known problem in the Linux kernel

[2] This is not the entire penalty suffered by an application. It is very difficult, in general, to quantify all the effects of context switches on contemporary microprocessors. For example, extra cache misses occur as a result of process state having been evicted from the cache when another process ran. These effects are not seen at context-switch time but are manifest as extra cycles consumed during instruction execution long after the context switch. The cache effects continue until the working set is once again loaded.

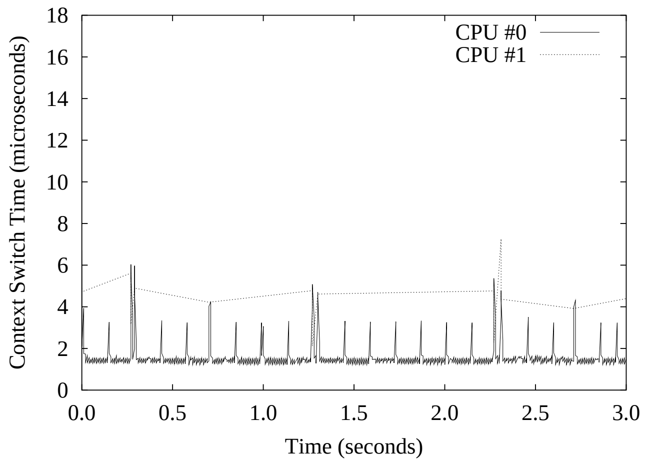

Fig. 12. Duration of Context Switches on an “Idle” Machine

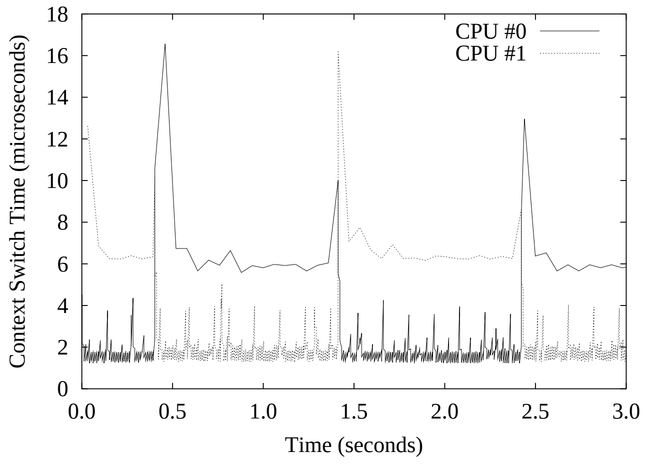

Fig. 13. Duration of Context Switches with a CPU-bound Process

scheduler which causes a process to migrate to idle processors in a round-robin fashion [15]. It is one of the items being worked on in the experimental 2.5 kernel. The spikes in context switch duration immediately after a process migrates to the other CPU are due to the cache effects mentioned earlier.

V. Future Work

Our implementation of MAGNeT can be improved in several ways. We would like to allow the user to set various MAGNeT parameters (e.g., the kinds of events to be recorded, the size of the kernel buffer, etc.) at run-time rather than at kernel compile-time. Allowing run-time user configuration of the MAGNeT toolkit could be accomplished by making the current `/proc` file writable. Run-time configuration would greatly increase the usability and flexibility of the MAGNeT toolkit.

Another potential area of improvement in MAGNeT is the mechanism used to store recorded data from the kernel buffer to disk. Rather then have a user-level process, a better approach may be to utilize kernel threads to perform all steps of the instrumentation [16]. With this methodology, the need for the special device file, the file created by `mkmagnet`, and the kernel/user shared memory would be eliminated. In addition, kernel threads may lower MAGNeT’s already low event loss rate by eliminating the overhead of `magnet-read`. However, implementing kernel threads may impede exporting real-time data to network-aware applications. The use of kernel threads may be explored for future versions of MAGNeT.

Timing with CPU cycle counters can be problematic on contemporary CPUs which may change their clock rate according to power management policies. If the kernel detects such changes, MAGNeT could easily hook into the clock-rate detection code and output new `MAGNET_SYSINFO` events. These events, containing new timing information, would allow correct post-processing in spite of CPU clock-rate changes. However, current Linux production kernels are unable to detect CPU clock rate changes at run-time. We will modify MAGNeT to account for CPU clock-rate changes when the Linux kernel provides a mechanism for doing so.

For applications which are compiled to use system shared libraries (rather than statically-compiled libraries), an alternative method of gathering application traffic patterns is to provide a shared library which instruments network calls before passing the calls on to the original system library. Since MAGNeT records network call events only within the kernel, the use of such an instrumented library (which records application events in user space, before any context switch) is complimentary to the approach taken in MAGNeT. Using such a library in conjunction with MAGNeT would allow system call overhead to be quantified while still requiring no change to applications.

VI. Conclusion

Current traffic libraries, network traces, and network models are based on measurements made by `tcpdump`-like tools These tools do *not* capture an application’s true traffic demands; instead they capture an application’s demands *after* having been modulated by the protocol stack. Therefore, existing traffic libraries, network traces, and network models cannot provide protocol-independent insight into the actual traffic patterns of an application. The MAGNeT toolkit fills the void by providing a flexible and low-overhead infrastructure for monitoring network traffic anywhere in the protocol stack.

ACKNOWLEDGEMENTS

This work was funded in part by the U.S. Department of Energy Laboratory Directed Research program and the U.S. Department of Energy Office of Science.

REFERENCES

[1] P. Danzig and S. Jamin, “tcplib: A Library of TCP Internetwork Traffi c Characteristics,” `http://irl.eecs.umich.edu/jamin/papers/tcplib/tcplibtr.ps.Z`, 1991.
[2] “The Internet Traffi c Archive,” `http://ita.ee.lbl.gov/html/traces.html`.
[3] A. Kato, J. Murai, and S. Katsuno, “An Internet Traffi c Data Repository: The Architecture and the Design Policy,” in *INET’99 Proceedings*.
[4] V. Paxson and S. Floyd, “Wide-Area Traffi c: The Failure of Poisson Modeling,” *IEEE/ACM Transactions on Networking*, vol. 3, no. 3, pp. 226–244, June 1995.
[5] “`tcpdump`,” `http://www.tcpdump.org`.
[6] W. Matthews and L. Cottrell, “The PingER Project: Active Internet Performance Monitoring for the HENP Community,” *IEEE Communications*, May 2000.
[7] A.J. McGregor, H-W Braun, and J.A. Brown, “The NLANR Network Analysis Infrastructure,” *IEEE Communications*, May 2000.
[8] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis, “An architecture for large-scale internet measurement,” *IEEE Communications*, 1998.
[9] CAIDA, “CoralReef Software Suite,” `http://www.caida.org/tools/measurement/coralreef`.
[10] J. Semke, “PSC TCP Kernel Monitor,” Tech. Rep. CMU-PSC-TR-2000-0001, PSC/CMU, May 2000.
[11] B. W. Tierney *et al.*, “The netlogger methodology for high performance distributed systems performance analysis,” in *Proceedings of IEEE the High Performance Distributed Computing Conference (HPDC-7)*, Jul 1998.
[12] J. Hay, W. Feng, and M. Gardner, “Capturing network traffi c with a magnet,” in *Proceedings of the 5th Annual Linux Showcase and Conference (ALS’01)*, Nov 2001.
[13] “Netperf,” `http://www.netperf.org`.
[14] W. Feng, J. Hay, and M. Gardner, “Magnet: Monitor for application-generated network traffi c,” in *Proceedings of the 10th International Conference on Computer Communication and Networking (IC3N’01)*, Oct 2001.
[15] M. Kravetz, “Cpu affi nity and ipi latency,” Jul 2001, `http://www.uwsg.indiana.edu/hypermail/linux/kernel/0107.1/0770.html`.
[16] Andreas Arcangeli, Private Communication.

APPENDIX

The following optional data structure can be compiled into MAGNeT to export TCP- and IP-specific protocol state to user-space.

```
struct magnet_tcp {
  /* data from "struct tcp_opt" in
     include/net/sock.h */

  /* TCP source port */
  unsigned short  source;
  /* TCP destination port */
  unsigned short  dest;

  /* Expected receiver window */
  unsigned long snd_wnd;

  /* smothed round trip time << 3 */
  unsigned long srtt;
  /* retransmit timeout */
  unsigned long rto;

  /* Packets which are "in flight" */
  unsigned long packets_out;
  /* Retransmitted packets out */
  unsigned long retrans_out;

  /* Slow start size threshold */
  unsigned long snd_ssthresh;
  /* Sending congestion window */
  unsigned long snd_cwnd;

  /* Current receiver window */
  unsigned long rcv_wnd;
  /* Tail+1 of data in send buffer */
  unsigned long write_seq;
  /* Head of yet unread data */
  unsigned long copied_seq;

  /* TCP flags*/
  unsigned short  fin:1,syn:1,rst:1,
      psh:1,ack:1,urg:1,ece:1,cwr:1;
}; /* struct magnet_tcp */

struct magnet_ip {
/* IP header info */
  unsigned char  version;
  unsigned char  tos;
  unsigned short id;
  unsigned short frag_off;
  unsigned char  ttl;
  unsigned char  protocol;
}; /* struct magnet_ip */
```

Fig. 14. MAGNeT Extended Data for TCP and IP