Architecture of a Network Monitor

Andrew Moore, James Hall, Christian Kreibich, Euan Harris, and Ian Pratt
University of Cambridge Computer Laboratory
JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom
{firstname.lastname}@cl.cam.ac.uk

Abstract— **This paper describes a system for simultaneously monitoring multiple protocols. It performs full line-rate capture and implements on-line analysis and compression to record interesting data without loss of information. We accept that the balance must be maintained in such a system between disk-bandwidth, CPU-capacity and data-reduction in order to perform monitoring at full line-rate. We present the architecture in detail and measure the performance of our sample implementation, Nprobe.**

Index Terms— **Passive network monitoring, full line-rate capture, multi-protocol analysis**

I. Introduction

CURRENT technologies for the passive monitoring of networks make limited use of the state-information present as part of each TCP/IP flow. While Intrusion Detection Systems (IDSs) often perform pattern matching from reconstructed data-flows, the processing (and modeling) of state in real-time has been rare. By using the state information of TCP/IP and higher-level applications it becomes possible to infer network properties, such as loss [1] and round-trip time [2], thereby disentangling the characteristics of network and transport layer from the end-system applications.

We present a network monitoring architecture currently able to interface with 1 Gbps full-duplex networks. Our approach is to perform multi-protocol analysis; this requires capture of all data from the line. The integrated analysis of application, transport and network protocols allows their interactions to be studied. Additionally, the association of transport and network state allows for data-reduction, transforming a *capture-to-disk* problem into a tractable trade-off between on-line processing and disk-bandwidth. This system has been successfully operated on carrier-grade networks and, using only commodity hardware, has the potential for economical mass deployment.

A. Related Work

Passive network monitoring has seen a number of previous approaches. An evolution of approaches has been spawned by the growth in network speeds. Examples of monitoring include those based upon kernel-implementations of packet capture, such as packetfilter [3] and *tcpdump/libpcap* [4]. Specialist hardware has seen important application in work such as OC3MON [5] as well as more recently with DAG [6] – with demonstrated part-packet capture rates of up to 10 Gbps. Additionally, new projects in this area are getting underway: the European SCAMPI [7], an architectural design for data capture; and MAGNeT [8]. MAGNeT is a monitoring mechanism for exporting kernel events to user space. That project uses instrumentation of the kernel/transport stack to provide an insight into network characteristics and application behavior. However, we expect that systems such as MAGNeT will be limited in deployment through the need to instrument an operating system.

Aside from our approach, other monitor systems that allow multi-protocol analysis are `Windmill` [9] and `BLT` [10]. `Windmill` is a probe architecture designed to reconstruct application level protocols and relate them to the underlying network protocols. `BLT` [10] extracts full TCP and partial HTTP level packet traces. In contrast to our design, `BLT` uses tape storage while `Windmill` exports data during collection. Such approaches have trade-offs (complexity and limited capture rates) but do allow continuous collection of trace information. A design constraint of our architecture from the outset is disk capacity (although this is alleviated by high data reduction rates). The provision of disk capacity sufficient to record twenty-four hour traces (hence any diurnal traffic cycles) is likely to cater adequately for the majority of studies, and longer periodic variations in traffic can be handled by repeated traces. Additionally, it is common carrier practice to operate transmission capacity at substantially lower than maximum rate [11], this means that while our system must handle full-rate traffic bursts, over the course of long capture runs, full-rate capture does not mean total line-rate capture.

Along with `Windmill` and `BLT`, our approach incorporates a *packet filter* scheme. Both `Windmill` and `BLT` implement coarse-grained data discard through

kernel/application-based packet filtering, but in contrast, our filter architecture is relatively simple and serves a different purpose – to support *scalability* as described in Section II-D.

IDSs such as SNORT [12] can also perform on-line capture and analysis. Modules in SNORT (e.g. `stream4`) allow the inspection of packet state and subsequent reassembly of the TCP streams; the re-assembly module has a theoretical capability to handle up to 64,000 simultaneous streams. However, an IDS based upon signature/pattern matching is a very different problem to that of a multi-protocol analyzer. Our approach is a tool that performs analysis of TCP/IP and application protocols, this includes providing timing relationships between events; SNORT and other IDS have the objective of performing fast string-matching against all packets – capturing only those few packets of interest.

The work we present here is drawn upon a prototype presented in [2], called Nprobe. This monitor has been used in a number of studies, the most recent being a study of web traffic [13].

The remainder of this paper is structured as follows: Section II presents our probe architecture in detail. Section III describes our testing methodology, traffic preparation and hardware setup, followed by our performance evaluation. Finally, Section IV discusses our results, and Section V summarizes the paper.

II. Architecture

A. Motivation

A constraint upon our design from the outset has been a desire to capture as much data, from as many different levels of the network stack: network, transport and application, as practical. For a low-speed network, our solution could have been a *capture-to-disk*, consisting of a `tcpdump`-type program combined with a significant quantity of off-line processing. However, limits of bandwidth to disk and a need for improved time-stamping combined with substantial scope for data-reduction and the increasing speed of the general-purpose CPU prompted an alternative approach.

In order to capture all data at network, transport and application level, our approach incorporates several components. First is the recognition that there is considerable redundancy in network data: both within any network packet and between network-packets. Application of simple, loss-less compression systems have long been used to improve the performance of link-layer mechanisms such as high-speed modems – ample scope exists for compressing captured data in the same manner. A clear trading relationship exists between the amount of CPU required (proportional to the degree of compression) and the bandwidth to disk. It is possible to consume all available disk and CPU resources with a sufficiently large quantity of data, although for more modest requirements the limiting factor tends toward the PCI bus bandwidth.

Given that discard is the best form of compression, considerably more compression can be achieved by performing an extraction that captures only significant or interesting information. In this way the capture of web traffic would only involve recording the minimum of TCP or IP header information for each HTTP transaction along with the HTTP transaction itself. Even the data of a retrieved object is not relevant and discarded, as it is the protocol operations that are of interest. A significant margin of compression can be wrought using this technique.

Clearly, in exchange for raw bandwidth to disk our architecture must be able to recognize packets that belong to each flow of interest (in order to perform compression upon them). An architecture that performs real-time protocol-feature extraction from reassembled TCP flows, on a carrier scale, is considered to be the central contribution of this work.

In addition to such application-specific compression, our approach includes a method of bandwidth splitting between multiple machines in order to share workload.

For a single monitor, the bandwidth to disk and available CPU will ultimately prove to be a bottleneck. For the load-splitting approach we propose there is a hard upper-limit on the amount of network traffic any particular IP host-host pair is responsible for – this will be the bandwidth able to be managed by a single monitor. For the wider public Internet as well as common research and academic networks we consider this an acceptable design decision. This imposition does mean our system is not suited to the more unusual situations such as the 10 Gbps networks carrying only a single traffic flow [14].

In the worst-case splitting approach, traffic from a single IP host-host pair may be striped among a number of monitor machines. This approach reduces to a *capture-to-disk* problem as only off-line analysis is possible of the network, transport and application flows. For our approach we use an XOR'd product of the two addresses of the IP host-host pair as the input to the filter of traffic for each monitor. The filtering is performed by the network interface thus the discarded packets never cause load on the monitor's CPU or the PCI or disk buses.

This approach means that each monitor will see all traffic in any particular flow and thus can subject them

to the network, transport and application compression mentioned above. Although rare, several deployments we have made required some load balancing among monitoring machines when unusual quantities of traffic are detected between particularly well-connected hosts. This process is easily detected as resource-overrun on a monitor and the solution is to compute a more appropriate filter-table for the monitors. Although IP addresses are reallocated, making this process iterative, we have noted that such iteration is on a very long time-scale (months-years) that matches the commissioning time-scale of new servers and routers.

B. Architectural Overview

Figure 1 illustrates the principal components and arrangement of our architecture.

Our implementation is built upon the GNU/Linux operating system, modified to improve the behavior of the asynchronous `msync()` system call used in writing data log files to disk. We also modify the firmware of the network interface card (NIC) to provide the high-resolution time stamps described in Section II-C.1 and packet filtering ability described in Section II-D.

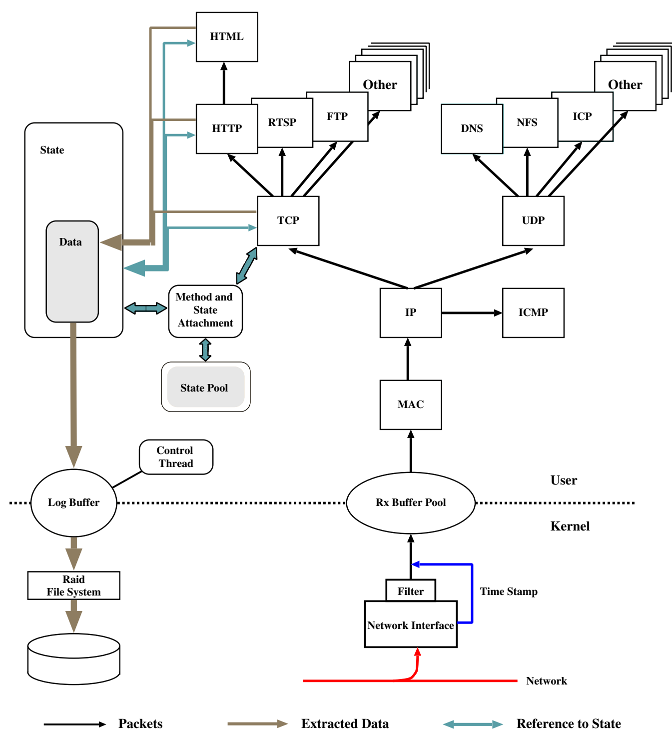

Fig. 1. The principal components of the architecture

Our approach includes three stages: packet *capture*, packet *processing* and *data storage*. Buffering between the stages accommodates burstiness in packet arrivals and variations in packet processing and data storage rates.

Packets arriving from the network are presented to a simple filter we place in the NIC's firmware; those passing the filter are time-stamped and transferred into a kernel memory receive buffer pool without any further processing.

A monitoring machine has one or more receive buffer pools, each associated with one user-level process and mapped into its address space. Each process presents packets held in the receive buffer pool to a series of protocol-based modules which extract the required data from each packet *in-situ*. Once all modules have finished with a packet, the buffer is returned for re-use by the network interface driver. Modules are generally designed such that they process and return buffers in FIFO order, but sometimes it is useful to hold on to a buffer, for example when the HTTP module is parsing a flow that has been re-ordered due to loss. Such 'held buffers' can be quickly reclaimed by the system in an least-recently-allocated fashion if the number of available network buffers drops below a threshold.

Data extraction is normally dependent on the context provided by the processing of preceding packets. This context is held as state which is attached to each packet as it is processed. Extracted data is temporarily stored as part of the attached state.

When the last packet of an associated series has been processed, all of the extracted data held in state is copied into an output log buffer from where, once sufficient data has been accumulated, it is written to a large RAID disk.

The architecture employs protocol-specific modules which in turn define which data should be stored and which data ought to be discarded. Thus for the HTTP module, the principle data-compression is achieved by the discard of the data objects. The module fingerprints each object using a CRC64 hash while the packets are in memory; however the objects themselves are not saved. The hashes allow us to recognise references to the same data object even if they use different URLs.

C. Capture

There are two approaches to providing data for the processing system. The first is to capture the data on-line: the data is provided directly to the processing system from the wire. This is the *standard* mode of operation. The second approach is to provide the processing system with a trace captured off-line.

1) On-line capture: In this mode, the modified NIC firmware prepends each accepted packet with an arrival time stamp generated by a clock on the card. In this way inaccuracy due to latency between packet arrival and processing is eliminated.

The clock provided by the NIC currently used by our implementation provides timing with an accuracy and precision of approximately 1 millisecond although this is vulnerable to temperature drift. As packets are processed, the NIC-generated time stamps are periodically compared with the system clock; the current NIC clock frequency is calculated from the elapsed time of both, and its current offset from real-time noted. As packets are drawn from the receive buffer pool these two parameters are used to calculate an accurate real-time arrival stamp for each. The periodic comparison between NIC-generated time stamps and the system clock is based upon a small number of repeated readings of both in order to identify and eliminate inaccuracies which may arise as a result of intervening interrupt handling.

Time stamps generated in this way have a relative accuracy of one or two microseconds, and an absolute accuracy determined by the system clock's accuracy – typically within a few milliseconds, using the Network Time Protocol [15]. While such precision is not sufficient to accurately measure the serialization times of back-to-back packets at network bandwidths of 1 Gbps and above, it is of the same order as the serialization times of minimum-size packets at bandwidths of 100 Mbps or small (512 octet) packets at bandwidths of 1 Gbps.

2) Off-line file input: In this mode of operation, the processing system can, as an alternative to drawing packets from the receive buffer pool, read them from a `tcpdump`-format trace file. This facility was provided for development purposes, but it is also useful to have the ability to apply data extraction and analysis to traces collected using `tcpdump` (although, of course, the limitations of `tcpdump` will still apply).

The ability to read packets from `tcpdump` trace files is also used in investigating data processing failures in which case offending packets are dumped to an error log. The value of such a system is that any anomalies seen *in-situ* may be analyzed at leisure; the exceptions having caused automatic generation of packet-traces that then may be used as input in a development feedback loop to program code to cope with the observed exceptions.

The `tcpdump` format was chosen as it allows examination of the error log files using `tcpdump`, and in particular because the packet filter can then be used to select packets for display or extraction by error type (possibly in combination with other criteria).

D. Scalability

It must be recognized that the ability of a monitoring system to keep pace with packet arrivals will be insufficient at some point due to increasing traffic volumes.

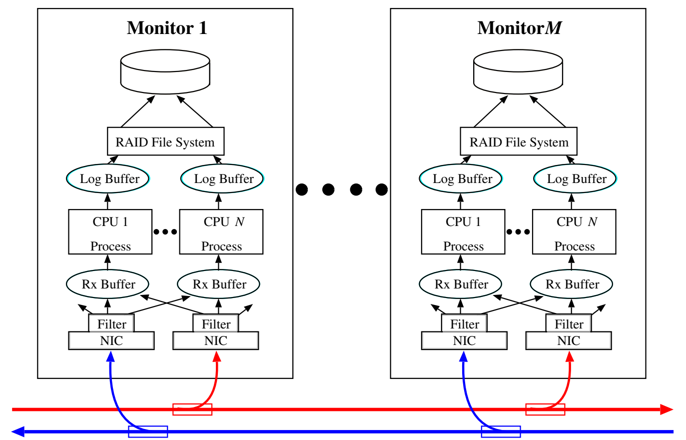

Fig. 2. System scalability

This applies equally to the available processor cycles, memory access times as well as bus and I/O bandwidth.

The scalability of our implementation is based upon the *striping* of packets across multiple processes, possibly running on multiple monitors; although the capacity of individual PC-based monitors may be less than that of purpose-designed hardware, their relatively low cost makes the employment of monitoring clusters an attractive proposition. Striping may alternatively be used as a form of coarse-grained data discard in order to collect a sample of the total network traffic by discarding a sub-set of the total stripes.

The filter employs an n-valued hash based upon packets' XOR'd source and destination IP addresses – hence distributing traffic amongst n processes, each dealing with a specific aggregated bi-directional sub-set of the total. This filter is implemented in the firmware of each of the monitor's NICs: accepted packets are transferred into the receive buffer pool associated with an individual process; rejected packets are dropped, hence not placing load on the monitor's PCI bus. Where multi-processor monitor machines are employed, one process is run per processor, thereby exploiting affinity of flows to processors. If the throughput required exceeds that possible on a single machine, multiple monitors may be deployed, forming a monitoring *cluster*.

A cluster using multiple CPUs per monitor and multiple monitors is illustrated in Figure 2. To achieve full analysis of the relationship between flows, additional off-line processing is required to merge the data captured by the multiple monitoring systems.

A limitation imposed by our approach is that the monitor cannot deal with traffic of any single host-host flow that exceeds its capacity. In justification of such a limitation our *public-carrier* deployment experience has involved a major UK ISP which provides dial-up and cable-modem access (e.g. 56kbps – 2Mbps). Due to

such a limit on the last-hop link capacity, an architecture that limits host-host data rates to being below that which any single monitor system can comfortably process was a quite reasonable design decision. Such a limit will preclude the monitor from more unusual situations, such as 10 Gbps networks carrying only a single traffic flow [14].

E. State and Data Management and Format

In our design, control is as far as possible *data driven* (e.g., packets are passed from one protocol processing module to another according to their content): the packet processing loop is driven by the presence of packets in the receive buffer pool, data copy from state to log buffer is triggered by the processing of the final packet of an associated sequence. No control processes are introduced (with one exception – the infrequently called file management thread), hence adding only a minimal overhead.

Our implementation avoids memory-copy penalties by using the kernel's memory-mapping facility to map the receive buffer pool directly into the user-level packet processing process(es)' address space and by memory mapping the output files. All data extraction and other processing is carried out on the packets *in-situ*.

We will now describe a number of the data structures used to implement our approach to convey how a carefully architected system is capable of simultaneously processing carrier-grade numbers of concurrent TCP/IP flows.

A *data association unit* (DAU) is a sequence of packets having semantic or functional continuity at one or more levels of the protocol stack, and from which data is extracted, aggregated and associated. Each current DAU has a corresponding *state storage unit* (SSU) for holding state information.

In this way state and data are identified, initialized and managed as a single unit, and data can be transferred efficiently as a single continuous block of memory.

The continual birth and death of DAUs gives rise to a very high turnover of associated SSUs. So we create a pool of the various SSU structures when the monitor is activated, and draw from it as required. The pool is organized as a stack so as to maximize locality of reference. As new DAUs (packet sequences) are encountered a new SSU is drawn from the pool and entered into a hash list based upon IP addresses and port numbers[1]; this provides efficient co-location of the SSU.

[1]The hashing strategy used is actually two-stage, based first upon IP addresses and then port numbers. The option to associate multiple connections as a single flow is thereby provided.

Flexibility in the degree and type of data collected from packets is reflected in the SSU. State and data vary with a studies' requirements within a specific protocol. Additionally, appropriate chaining of DAU and SSU allows the associations of data to be established, for example allowing speedy processing for multiple HTTP transactions over one TCP connection.

The size of data fields within an SSU may be statically or dynamically determined. Numeric or abstracted data will be stored in numerical data types of known size, but a mechanism for the storage of variable sized data (e.g., the alphanumeric `URL` field of an HTTP request header) is also required. The demands of efficient packet processing preclude an exact memory allocation for each data instance and a trade-off must be made between allocating sufficient space to accommodate large requirements and making heavy demands upon the available memory. A count of length is kept and used to ensure that only the data (as opposed to the size of statically allocated buffers) is logged.

Timing data is central to many studies and, as described in Section II-C.1, packet time stamps are provided with a precision of 1 μs. Time stamps recorded in a thirty two bit unsigned integer would overflow/wrapround in a period of a little over one hour. While such wrap-rounds may be inferred, significant time stamps (e.g., TCP connection open times) are recorded as sixty four bit quantities in epoch time.

F. Off-line Processing

The data produced by our implementation is collected into a pre-defined format; the format varies depending upon the data captured. In order that the output data can be easily manipulated, the storage formats are maintained using `SWIG` [16]. This creates interfaces that off-line processing systems may use to access the logged data structures.

The automatic compile-time creation of interfaces to the log files enables the rapid prototyping of programs able to retrieve data from the log files. The majority of our off-line processing is done using Python, a language suited to rapid prototyping. While Python is an interpreted language, any impact upon processing performance is minimal as its use is restricted to the off-line processing phase.

G. Protocol Modules

As part of the overview of the architecture, Figure 1 illustrated the principal components and arrangement. Notably, each protocol – at network, transport and application layers – makes up a separate module. In our

current implementation the majority of work has been in the IP, TCP, HTTP and HTML modules. Modules may be shared in their interconnection, for example, the IP and TCP modules are held in common across a large number of applications. We also have initial implementations of protocol modules for FTP and DNS; additionally an early version of an NFS module was described in a previous publication [17].

The addition of new protocol modules to the architecture centers upon the construction of a number of methods activated upon packet-arrival. The existing architecture's hierarchy allows easy additions to be made, a simple protocol module may cause matching packets to be logged, the bytes or packets counted, or the packets to be discarded outright. In contrast, more complex methods may elect to provide packet data to higher-level protocols (e.g., TCP flows providing data to HTTP) after processing (TCP performing data reassembly). A direct result of the logging data-structures being central to each design is that construction of a module will, through the SWIG interfaces, simultaneously construct the off-line data interfaces.

The result is that adding modules to the overall architecture is an incremental process, adding functionality only as it is required. Combined with the iterative debugging process described in Section II-C.2, the development of new module is quite straight-forward.

III. Testing and Results

In order to test our system reliably, we set up a network environment that allowed us to expose the probe to varying traffic patterns in a controlled manner, while at the same time enabling us to recreate realistic traffic characteristics.

A. Architectural Aspects

In order to get a clear idea of our implementation's performance, we created several test cases that specifically exercised the potential bottlenecks of Nprobe. Our evaluation covered the following aspects:

1) System Components: These tests evaluated the capacity of systems involved in moving the data from the network onto disk: the NIC hardware, the hardware and software involved in moving the data from the NIC to the Nprobe process, and the mechanisms for moving data onto the local disk. We had to make certain assumptions about the hierarchy of bottlenecks in our architecture – these assumptions were also imposed on us by the hardware we used.

2) Module Performance: Our evaluation of Nprobe's protocol modules has focused on the TCP/IP, HTTP and HTML modules. The work these modules perform can have significant impact on the system's performance. In addition to the TCP/IP checksumming, the HTTP interpreter computes a "fingerprint" in form of an MD5 hash for each HTML object – this allows individual objects to be identified regardless of the URL they are accessed by. Such operations are significant overheads on the total system and can be selectively tested by using purpose-built input traffic.

3) Total-system Performance: Finally, we investigated the overall system performance of our implementation by observing its behavior when exposing it to increasing volumes of a typical traffic mixture as observed on our laboratory's network. We measured the performance of the system with both IP and HTTP-only traffic.

B. Test Traffic Preparation

The basic input traffic we used for our experiments consisted of a full-packet capture of our laboratory's off-site traffic collected across several weekdays. Using this packet data, we generated sets of traffic to allow testing of specific bottlenecks in our implementation:

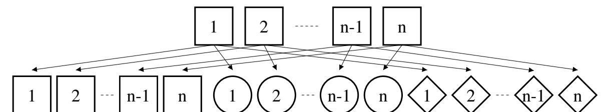

Fig. 3. Generating a sequence of individual HTTP transactions

- *Sequence of HTTP transactions, no overlap*
  We selected a small but complete HTTP transaction consisting of seven packets from the captured traffic and concatenated this transaction massively until the resulting trace reached a pre-determined size. Every instance of the connection was assigned different IP addresses, to make the requests appear to be individual instances. Figure 3 illustrates the process.

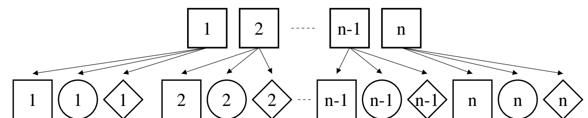

Fig. 4. Generating overlayed HTTP transactions

- *Sequence of HTTP transactions, overlayed*
  To test the probe's behavior when it has to keep state, we selected a large HTTP transaction that occurred in our traffic. To build up state, multiple instances of this connection were overlayed packet

by packet, each with unique IP addresses, resulting in up to 10,000 overlays. Figure 4 illustrates the process.

- *Repeatable IP traffic*
  To test the probe's general performance on a "normal" mixture of IP traffic, we also generated input traces that preserve realistic traffic characteristics, while eliminating unwanted side-effects when being replayed repeatedly. To do this, we demultiplexed the captured traffic on a per-host basis, looking only at traffic entering or leaving our lab. While doing this, we removed any incompletely captured TCP connections to avoid unrealistic state building up inside the probe (the probe is designed to time-out flow records when memory is low, but this rarely happens in normal operation, except when major network re-routing occurs). We then again created trace files out of this host-based traffic, sampling through the full range of available per-host traffic volumes.

Since our test setup required considerably more traffic than directly obtained using the methods described above, we duplicated the traces as often as necessary, moving the range of IP addresses in each clone to an individual range. While replaying, we thus avoided any artificial traffic duplication.

Most of the tools necessary for the traffic manipulations were written as plugins for our `netdude` [18] traffic editor, as it provides a higher-level API for these kinds of operations compared to handwritten libpcap applications.

C. Test Environment

The probe was tested against a traffic load generated by a cluster of PCs attached to a pair of high-performance Ethernet switches. This testbed could replay previously recorded real network traffic at a controllable rate from a few kilobits to several thousand megabits per second. Initial experience with the `tcpreplay` [19] utility indicated poor performance due to per-packet overheads; our solution was to create a replacement, `tcpfire` [20], that enabled line and near-line performance. Once prepared, the traffic files were distributed to machines that replayed this traffic to our testbed. While the hardware of each cluster machine is modest — each node generating 25–100 Mbps of sustained traffic — this traffic is multiplexed onto a fiber link, and using up to 40 source machines, data-rates of up to 1 Gbps can be achieved in both directions.

The probe was inserted into the fiber link between the two switches and, using an optical splitter, diverted a small percentage of the light of each fiber to a pair of NICs in the probe. Coarse-grained control of the traffic volume was obtained by adding and removing source nodes from the cluster; finer-grained control was possible through controls offered by the replay program running on each node.

Several traces were used to evaluate the performance of the network interface (SysKonnect SK-9843-SX) and our monitor system (Intel Xeon 2.4 GHz with 512 KB L2 in an Intel e7500-based motherboard, using 2 GB of DDR-SDRAM, 64 bit/66 MHz PCI bus, with 3ware 7850-series ATA-RAID card providing a hardware RAID 5 on top of 8 IBM Deskstar 120 GB 7200 RPM drives).

D. Results

As illustrated in Section III-A, each of our tests was designed to evaluate one particular aspect of the system's performance. The following sections describe the tests in detail and present the results of our measurements.

1) Component Performance: Several traces were used to evaluate the performance of the system hardware. Performance is determined by maximum data rates observed just prior to packet loss by the monitor. The results of Table I clearly illustrates that very small packets have a disastrous impact upon the system-performance. We noted that for a network interface we had used earlier (3Com 3c985B) not only was performance lower, but the card was a clear bottleneck for small packets. In contrast, for the SK-9843-SX, the bottleneck became the (large) number of interrupts generated on the PCI bus. While some interrupt coalescing is performed by our driver for this card, this is clearly an area for future work.

Although it is not a directly comparable system, we compared our implementation's performance to that of `tcpdump`, a system commonly used for capturing network traffic. Packets captured using `tcpdump` have to cross the kernel-user space boundary, using network code optimized for day-to-day operations of transmit and receive.

2) Module Performance: To test the CPU usage of the modules, we used the traces containing overlays of large HTTP transactions, as described in Section III-B. A number of these trace files were multiplexed together, increasing the quantity of work the CPU had to perform. Packet-loss based on CPU exhaustion occurred when the traffic load exceeded 790 Mbps (97.3 kpps) for a single monitor. By using a large transaction as input, the system bottleneck was moved from the network interface card and PCI bus into the CPU, since the transfer of a large data item is dominated by large packets. While this particular limit is solved with ever-faster CPUs as well

	64 byte	1520 byte	live mix
		Mbps (kpps)	
`tcpdump`– examine and discard	19.4 (36)	1151 (95)	209 (121)
`tcpdump`– capture to disk	16.6 (42)	339 (28)	192 (119)
Nprobe– examine and discard	95.9 (187)	1579 (130)	340 (217)
Nprobe– capture to disk	64.8 (127)	496 (41)	240 (154)

TABLE I

MAXIMUM TRAFFIC RATES FOR FIXED-SIZE PACKET TRACES AND A LIVE MIX REPRESENTING TYPICAL TRAFFIC. A SMALL VARIATION OF THE MEAN-PACKET-SIZE IS PRESENT IN THE LIVE TRAFFIC MIX.

as the use of multiple CPUs, single CPU performance provides an important base result.

Section II-E details the state and data management performed by our architecture. A clear restriction on the total amount of traffic processable is the number of TCP/IP and HTML/HTTP state engines required to exist simultaneously. This limits the number of concurrent TCP flows that may be reconstructed, and thus the number of HTML/HTTP flows that may be concurrently processed.

The limitation ought to be imposed as a hard-limit based upon the size of the state/data records and the total quantity of memory available in the monitor to create these state/data records. This is a test of our architecture's implementation and we can achieve 44000 flows at 304 Mbps (117 kpps). We anticipate this performance barrier may be imposed by the CPU-memory interface as it became clear that the system was not bound by disk or PCI bus speeds. Such conjecture may be proved by the use of a dual-processor system as improved L2 cache use would be anticipated to improve performance.

The second test of our architecture's implementation was the rate at which such state/data records can be released and reused: such functions are clearly important in *carrier-grade* monitoring that may observe hundreds of thousands of new flows per second. We tested this aspect using traces containing sequences of non-overlapping small HTTP transactions as described in Section III-B. Like the previous case each individual flow is between two unique hosts and as a result each trace file (of which up to 80 are used) represents near 2 million individual transactions among unique host pairs.

Our system was able to manage about 23500 transactions per second (at 54 Mbps (165 kpps)). Investigating our results, this value is due to other aspects of system performance (CPU-starvation) being the limit on performance and impacting the systems ability to manage and manipulate the large number of state/data records.

3) Total-system Performance: The final results we present describe the total-system performance when Nprobe is monitoring variable quantities of typical network traffic as well as just the flows carrying HTTP data. As mentioned above, the sample of typical traffic we used was recorded from the 1 Gbps link between a large academic/research organization and the Internet. While it is very difficult to compare performance for different traffic types, we have had extended experience with this system deployed in the national backbone of a major public UK ISP. That experience combined with other deployment experience at the department, institution and national-research network levels has provided us with sufficient confidence in the operation of our system.

The test trace used consisted of 88% TCP by packet (95% by byte volume). The TCP/IP traffic, in-turn consisted of 10% HTTP by packet, (54% by byte). At less than ten percent, this traffic is clearly not dominated by HTTP on a packet basis, however HTTP accounts for over half the total number of bytes transmitted.

Using the multiplexed traces of access traffic for a single research/academic department we find the probe is able to keep up with traffic intensities bursting at up to 500 Mbps measured over a 1 second interval. However, we note that the probe's sustainable link-rate is a slightly disappointing 280 Mbps.

The most significant cause of this performance limit is the trace's mean packet size, which is just 190 octets. Leading to a packet rate of 184 kpps, the per-packet overhead in both the analysis code and the network interface is detrimental to performance. On-going profiling and tuning work is attempting to address this bottleneck.

Experiments were also made using superficially similar traffic: the laboratory trace as used above was used as input, however a level of pre-processing removed all non-HTTP traffic. Thus, the difference now was that all traffic seen by the monitor consisted of 100% TCP/IP packets carrying HTTP transactions. The results: 189 Mbps (101 kpps) for a mean of 234 bytes/packet indicated that other parts of the implementation (aside from small-packet limitations) had come into play. Further investi-

gation indicated that the object fingerprint function was consuming most CPU. It would, however, be an error to indicate that the fingerprint function was dominating CPU, rather we consider that the combination of events — file system activity, stream-assembly and its data structure overheads, as well as the hashing functions — each made a contribution.

While we believe there is scope for more thorough optimization of our prototype implementation, these results are encouraging and validate the approach.

IV. Discussion

Section III-D.1 described experiments to illustrate limitations imposed by hardware or limitations that are fundamental to the current implementation. It is clear that one important area for future work is to optimize hardware and software behavior when faced with large numbers of small packets.

Another area of note was the limitation imposed by disk bandwidth and scope exists for testing of more exotic disk sub-system solutions.

The upper value we achieved using Nprobe for the largest packets (lowest per-packet overhead) without writing that data to disk indicates another issue to focus upon. We noted that the PCI bus was a common point of congestion in these tests. While the PCI bus bandwidth at 64 bit/66 MHz ought to have sufficient raw capacity a limit is imposed due to device contention (two network interfaces: one monitoring each direction) and due to bus transaction overheads. Scope exists to investigate still-faster PCI bus implementations, but also to explore unusual network-interface-card designs (e.g., multiple transceivers sharing the same PCI interface).

The tests of Section III-D.2 have illustrated the bounds imposed by our use of non-custom, commodity hardware. We do not suggest that the only solution is to use faster hardware; scope for code-optimization is also present. Furthermore, multiple-CPU motherboards are common-place and, while not presented here, the use of multiple CPUs (as described in Section II-D) is another avenue for investigation. However, our tests indicate that our current implementation is sound.

An interesting comparison may be made between the Nprobe performance when capturing all traffic to disk, and when performing as an analysis-compression system. Section III-D.3 describes system performance when analyzing and recording a typical traffic mixture. In comparison with the raw-throughput tests presented in Section III-D.1, 280 Mbps is a somewhat disappointing performance figure, but further tuning, and use of a cluster of machines would enable us to achieve our 2 Gbps target. While Nprobe in full-operation only provided a small gain (about 20 Mbps) over Nprobe operating in *capture-to-disk*, a substantial quantity of on-line processing was performed on the captured traffic, reducing both storage requirements and the off-line workload. One approach may be to reconsider the precise split in workload between on-line and off-line processing.

It has been a fundamental principle of this project to use only commodity hardware. Whereas in the short term this may have added to the difficulties inherent in monitoring high-speed networks, in the long term Nprobe is well-placed to reap the rewards of the continuous innovation and performance improvements which are the hallmarks of the commodity hardware industry. Particular examples are increases in CPU power, still governed by Moore's Law, and improvements in commodity NICs, which appear to follow a similar trend. However, even if these advances should prove inadequate, the Nprobe modular architecture is sufficiently flexible as to allow the incorporation of more exotic technologies, such as network processors. Offloading TCP reassembly and fingerprinting into these devices is particularly attractive.

Finally, while the ambition of this paper was to discuss the software and hardware architecture of a network monitor, the use of such systems encompass many real-world issues, among them: management and security. The architecture we outlined in this paper has the potential to make a full and complete record of a users network utilization, clearly there are both privacy issues for the user and security issues for data and monitor access. To this end, we have a monitor module that is able to provide crude anonymity through prefix-preserving re-mapping of IP addresses. We consider schemes of data management that offer accommodation of privacy, security and legal constraints an important aspect of future work.

V. Summary

We have presented an architecture developed to perform traffic capture and processing at full line-rate without packet loss. Performing on-line capture allows application-specific processing and application-specific compression. Combined with a powerful off-line processing interface, this approach can perform full capture to disk of interesting traffic features, yet remains capable of monitoring high-speed networks. Additionally, we demonstrated how this architecture achieves higher-speed monitoring through traffic separation and filtering.

Alongside the presentation of a network monitoring architecture, we documented a performance evaluation of our current implementation: Nprobe. Our implemen-

tation was shown to perform well, although several areas for future enhancements were also identified.

Future Work

Aside from studies of network traffic, future work for this project will concentrate on improving the implementation of our monitoring architecture. Particular emphasis will be on enhancing the application-processing systems — this will allow processing of other common network applications and thus allow the interaction of such applications with the network and transport protocols to be studied. Additionally, we are extending our system for 10 Gbps operation, and are currently planning test environments for its development and evaluation.

Acknowledgments

Specific thanks to Tim Granger, Ian Leslie, Derek McAuley, Martyn Johnson, Piete Brooks and Phil Cross. Thanks also to Intel Research Cambridge for providing access to network test facilities.

Finally, we acknowledge the valuable feedback provided by the anonymous reviewers and our colleagues at the University of Cambridge Computer Laboratory.

References

[1] J. Liu and M. Crovella, "Using Loss Pairs to Discover Network Properties," in *ACM Internet Measurement Workshop 2001 (IMW 2001)*, Nov. 2001.

[2] J. Hall, I. Pratt, and I. Leslie, "Non-intrusive estimation of web server delays," in *LCN 2001 The 26th Annual IEEE Conference on Local Computer Networks (LCN)*, Tampa, FL, March 2001.

[3] J. C. Mogul, "Efficient use of workstations for passive monitoring of local area networks," in *Proceedings of ACM SIGCOMM'90*, Philidelphia, PA, Sept. 1990.

[4] "tcpdump/libpcap," 2001, http://www.tcpdump.org/.

[5] J. Apisdorf, K. Claffy, K. Thompson, and R. Wilder., "Oc3mon: flexible, affordable, high performance statistics collection."

[6] I. D. Graham, S. Donnelly, J. M. S. Martin, and J. Cleary, "Nonintrusive and accurate measurement of unidirectional delay and delay variation on the internet," in *Proceedings of the INET'98 Conference*, July 1998.

[7] *SCAMPI, a Scalable Monitoring Platform for the Internet*, Lieden University (in collaboration), Mar. 2002, http://www.ist-scampi.org.

[8] M. K. Gardner, W. Feng, and J. R. Hay, "Monitoring Protocol Traffic with a MAGNeT," in *Passive & Active Measurement Workshop*, Fort Collins, Colorado, 3 2002.

[9] G. R. Malan and F. Jahanian, "An extensible probe architecture for network protocol performance meas urement," in *Proceedings of the ACM SIGCOMM '98 conference on Applications, tec hnologies, architectures, and protocols for computer communication*. ACM Press, 1998, pp. 215–227.

[10] A. Feldmann, "BLT: Bi-layer tracing of HTTP and TCP/IP," *WWW9 / Computer Networks*, vol. 33, no. 1-6, pp. 321–335, 2000.

[11] K. Papagiannaki, S. Moon, C. Fraleigh, P. Thiran, F. Tobagi, and C. Diot, "Analysis of Measured Single-Hop Delay from an Operational Backbone Network," in *Proceedings of IEEE INFOCOM 2002*, New York, NY, June 2002.

[12] Martin Roesch and Chris Green, "SNORT: The Open Source Network Intrusion Detection System 1.9.1," Nov. 2002, http://www.snort.org/.

[13] J. Hall, I. L. Ian Pratt, and A. Moore, "The Effect of Early Packet Loss on Web Page Download Times," in *Passive & Active Measurement Workshop 2003 (PAM2003)*, Apr. 2003.

[14] L. Cottrell, "TCP Stack Measurements," 2003, http://www-iepm.slac.stanford.edu/monitoring/bulk/fast/.

[15] D. L. Mills, "RFC 1305: Network time protocol version 3)," Mar. 1992.

[16] "Simplified Wrapper and Interface Generator," Jan. 2003, http://www.swig.org/.

[17] J. Hall, R. Sabatino, S. Crosby, I. Leslie, and R. Black, "A Comparative Study of High Speed Networks," in *proceedings of the 12th UK Computer and Telecommunications Performance Engineering Workshop*, September 1996, pp. 1–16.

[18] "Netdude, the NETwork DUmp Data Editor," 2003, http://netdude.sf.net/.

[19] "tcpreplay," 2003, http://tcpreplay.sf.net.

[20] "tcpfire," 2003, http://www.nprobe.org/tools/.