SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2004; **34**:47–67 (DOI: 10.1002/spe.557)

An extensible probe architecture for network protocol performance measurement

David Watson[1,*,†], G. Robert Malan[2] and Farnam Jahanian[1]

[1]*Department of Electrical Engineering and Computer Science, University of Michigan, 1301 Beal Ave., Ann Arbor, MI 48109-2122, U.S.A.*
[2]*Arbor Networks, 625 E. Liberty Street, Ann Arbor, MI 48104, U.S.A.*

SUMMARY

This paper describes the architecture, implementation, and application of Windmill, a passive network protocol performance measurement tool. Windmill enables experimenters to measure a broad range of protocol performance metrics both by reconstructing application-level network protocols and by exposing the underlying protocol layers' events. Windmill is split into three functional components: a dynamically compiled Windmill Protocol Filter (WPF), a set of abstract protocol modules, and an extensible experiment engine. To demonstrate Windmill's utility, we present the results from several experiments. The first set of experiments validates a possible cause for the correlation between Internet routing instability and network bandwidth usage. The second set of experiments highlights Windmill's ability to act as a driver for a complementary active Internet measurement infrastructure, its ability to perform online data reduction, and the non-intrusive measurement of a closed system. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: passive measurement; packet classification; packet filter; network protocols; Windmill

INTRODUCTION

The Internet has grown significantly, and its growth shows little sign of slowing. Understanding the interaction among the many Internet protocols is a key challenge necessary for its rational growth. As a real-world system with economic incentives for uptime and robustness increasing, it is difficult to take portions of the network down for measurement and testing. Moreover, the rate of growth for the Internet has placed a severe tax on the network infrastructure, leaving many resources such as

*Correspondence to: David Watson, Department of Electrical Engineering and Computer Science, University of Michigan, 1301 Beal Ave., Ann Arbor, MI 48109-2122, U.S.A.
†E-mail: dwatson@umich.edu

Contract/grant sponsor: National Science Foundation; contract/grant number: NCR-9710176
Contract/grant sponsor: DARPA; contract/grant number: F001670

Received 2 January 2001
Revised 20 May 2003
Accepted 20 May 2003

Published online 26 November 2003

routers and highly trafficked Web servers in a state of constant overload. Compounding the problem is that most of the software executing the protocols is *shrink-wrapped* and is not amenable to scrutiny or modification for performance measurement—a backbone router collapses within seconds with full debugging turned on. It is precisely at these points where the performance effects of protocol interaction are the greatest, and most poorly understood. This paper presents an architecture for an extensible software probe that can measure precisely these types of interactions under real-world conditions.

The software probe described in this work utilizes passive techniques for eavesdropping on target protocols. Groups of these point probes can be distributed throughout a target network to measure an aggregate performance profile of target protocols. Care has been taken during the probe's design to enable its placement in high bandwidth monitoring points. This allows the measurement of Internet protocols across a spectrum of vantage points, from routing exchange points and enterprise gateways to local area networks. There are many research groups involved in the deployment of Internet probe machines for the measurement of Internet paths and topologies, including the NIMI [1], Surveyor [2], and IPMA [3] projects. These projects have approached Internet measurement by utilizing active performance metrics [4]—measurements that perturb the network—such as one-way loss along a datagram stream, or periodic traceroutes between the probe and specific end-hosts. Our work complements these efforts. We have designed and implemented the architecture of a passive performance probe that can be used in conjunction with active probes—possibly housed on the same host—to measure and infer performance data from the underlying network flows without perturbing the network or infrastructure. Our probe architecture allows the measurement of high-level protocols, such as BGP, DNS, and HTTP, that sit above the Internet's base protocols. Our tool measures these protocols without modifying either the infrastructure—the routers, nameservers, Web caches, end hosts, etc.—or the host implementation of these protocols.

The architecture of our tool, Windmill, supports the passive performance measurement of application-level protocols through the use of protocol reconstruction and abstraction-breaching protocol monitoring. A probe experiment infers the end-host's view of a target protocol by recursively executing the lower protocol layers against the stream of incoming packets. Effectively, the probe reconstructs the view of the end hosts by passively monitoring the protocol's network frames. The experiments utilize interfaces exported by the probe to 'lift the hood' on the lower-layer protocols, violating their abstractions to examine events and data structures that are normally hidden from the higher layers. Together, these features allow an experiment to correlate lower-layer protocol events—including checksum and length errors, packet reorderings or retransmissions, and round trip estimates—with the behavior and performance of the reconstructed application-level protocol.

To accommodate both performance and extensibility, Windmill's software was split into three functional components: a dynamically compiled protocol filter, a set of abstract protocol modules, and an extensible experiment engine. Packet throughput is maximized through the use of a custom protocol filter which dynamically compiles and downloads native code into the kernel for fast multi-destination packet matching. For performance, the bulk of the user-level code is contained in a set of abstract protocol modules. These modules are C implementations of the base Internet protocols. Those protocol layers that do not change—such as IP, UDP, TCP, BGP, and HTTP—are implemented as abstract protocol modules. By calling these modules an experiment can efficiently execute a target protocol's stack on incoming matching packets. The probe's extensibility comes from the use of a custom dynamic loader that is used to load and manage the probe's experiments.

The main contributions of the Windmill probe architecture are as follows.

- *The implementation of our passive probe architecture.* Our current implementation is built using an off-the-shelf hardware and software base. This implementation utilizes both recursive protocol reconstruction as well as abstraction violation to measure application-level Internet protocols, such as BGP, HTTP, and DNS.
- *Attention to the intrinsic tradeoff between performance and extensibility by splitting the code into two pieces.* The performance critical code—for protocol reconstruction and memory management—was placed in the tool's libraries, whereas the extensibility support was constrained to a custom runtime library. Together these pieces enable dynamic experiments to be loaded, managed, and modified over long periods of probe uptime, while allowing for the high performance protocol processing necessary for high bandwidth vantage points.
- *Creation of the fast Windmill Protocol Filter (WPF).* Since the probe is designed to execute multiple experiments simultaneously, there is the possibility that several experiments may subscribe to overlapping packet flows. In order to make multiple experiment matching as fast as possible, this functionality was pushed into a custom packet filter. This filter utilizes dynamic compilation in conjunction with a fast matching algorithm to enable *one-to-many* packet *multiplexing* in a running time linear in the number of comparable fields for common cases. This is the same time complexity as the best most-specific *one-to-one* matching algorithms. Moreover, WPF addresses some fundamental limitations in past packet filtering technology—filters that demultiplex packets to endpoints by *most-specific* matches—by correctly handling ambiguous (overlapping) filters that do not have any natural or explicit ordering.
- *Investigation of an Internet routing instability conjecture.* The probe was used in an experiment designed to monitor and measure the BGP routing traffic exchanged between two peer border routers in order to validate one of the key observations presented in [5]. Specifically, the experiment provides a possible answer for the correlation between Internet routing instability and network bandwidth usage. The experiment suggests that the BGP protocol be modified for use with a UDP keepalive protocol.
- *A study of an Internet Collaboratory.* Windmill was used to measure the Upper Atmospheric Research Collaboratory. The experiment demonstrated the use of the tool on a real system that could not be modified for direct measurement, the use of the tool for online data reduction, and the power of using our passive tool to drive an active measurement apparatus.

RELATED WORK

Windmill differs from existing techniques in three main areas. First, Windmill presents a comprehensive architecture for passively monitoring network protocols. Second, Windmill uses a novel packet filter designed to provide fast, one-to-many packet multiplexing. Third, the combination of these features allows Windmill to monitor the behavior of shrink-wrapped systems without modification. Passive techniques have been used in many low-level protocol performance evaluation and modeling studies (examples include [6–8]). Past work has mainly addressed aggregate traffic characteristics at or below the TCP layer, either from an end-host perspective or as an observer at an intermediate node. Our work is targeted at the passive measurement of higher application-level protocol layers, and the correlation of lower-layer protocol events with their performance.

Many tools exist to collect low-level protocol information including `tcpdump` [9] and CoralReef [10]. Both `tcpdump` and CoralReef acquire network frames from an underlying data source and can either store those frames or provide further post analysis. In the case of `tcpdump` the analysis is limited to interpretation and display of network protocol structures. CoralReef, however, can support a wider range data sources and presents a uniform interface for developing higher level analysis. In contrast, Windmill provides a mechanism for several experiments to cooperatively share data from a single source and also provides built in support for monitoring high-level protocol information. More recent work, such as FLAME [11], extends the idea of a general purpose monitoring architecture to include the notion of safety. This extends the applicability of tools such as Windmill to domains where monitoring can be performed by untrusted clients.

There has also been significant work done on packet filters. Traditionally, they have used interpreted code [12–14] or the traversal of high-level data structures [15] for safety and portability; however Engler and Kaashoek's DPF (Dynamic Packet Filter) [16] uses dynamic code generation techniques for high performance packet demultiplexing. One of Windmill's components, the WPF, borrows from DPF in that it utilizes dynamic code generation for fast packet matching; however it differs from all other previous work in that it is a packet multiplexor (*one-to-many)*. That is, it multiplexes incoming packets to multiple concurrent experiments. Moreover, the multiplexing operation is done in the same time as a single DPF match. In addition, WPF addresses some of DPF's fundamental limitations in regard to ambiguously specified filters.

Measuring distributed systems and their protocols at the end host can be done by instrumenting the code directly or indirectly through profiling and simulation. Both approaches require access to the application's data structures and a deep familiarity with the code regardless of whether it is source or binary. This is not applicable to two types of systems: shrinkwrapped software, and production systems. Methods for performing continuous system profiling with low overhead are presented in [17,18]. While the accounting of time in a system is useful for the optimization of the end-host's software, it is not an effective mechanism for profiling the semantic performance of an application. An alternative method for measuring the system is to simulate it entirely [19]; however, this is not practical for measuring production systems where real-time events drive the protocols of interest. Compounding the problem of end-host measurement of any kind is the distortion, or perturbation, of the machine by the measurement apparatus. This problem is exacerbated when the host is overloaded. In contrast, our passive approach neither significantly perturbs the measured system nor requires knowledge of a protocol implementation's data structures and code layout.

ARCHITECTURE

Windmill's architecture consists of three functional components: a dynamically generated protocol filter; a set of abstract protocol modules; and an extensible experiment engine. The organization of these components is shown in Figure 1. The WPF matches the underlying network traffic against a dynamically compiled filter. This filter is constructed from the set of outstanding packet subscriptions from the engine's concurrent experiments. These individual subscriptions act as a filter for each experiment—they describe which packets the experiment is interested in receiving. The abstract protocol modules provide both efficient implementations of target protocol layers and interfaces for accessing normally hidden protocol events. These modules are composed to enable the efficient

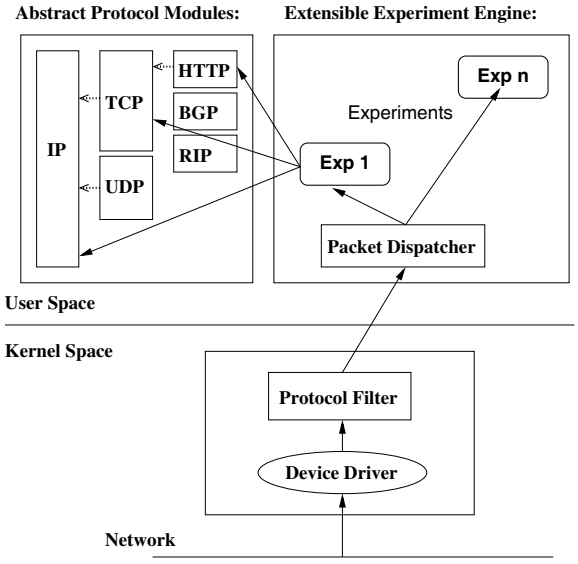

Figure 1. Organization of Windmill's architecture.

execution of the protocol stack on incoming packets. The extensible experiment engine provides an infrastructure for the loading, modification, and execution of probe experiments. Additionally, the experiment engine provides interfaces for the storage and dissemination of experimental results.

Experiments are loaded into the experiment engine through an interface controlled remotely by the probe's administrator. Once installed in the engine, an experiment subscribes to packet streams using the abstract protocol modules. These subscriptions are passed to the protocol filter which dynamically recompiles its set of subscriptions into a single block of native machine code. This code is installed in the kernel for fast matching and multiplexing of the underlying network traffic. Upon a packet match the protocol filter sends the packet along with the set of matching experiments to the packet dispatch routine in the experiment engine. The packets are then given to each matching experiment. Each experiment then uses the abstract protocol modules to process the standard protocol information in the packet. Rather than starting with the lowest layer, usually IP, the packet is given to the highest level protocol module, e.g. HTTP or TCP. These higher level protocol modules then recursively call the lower layers of the protocol stack. Afterwards, the experiment can extract the results of the packet's processing from any of the protocol layers. These results include the protocol frame or byte-stream service exported by the lower layers, or protocol events and error conditions triggered by the packet.

The current implementation of Windmill is based on off-the-shelf software and hardware. A custom version of the FreeBSD 3.3 kernel serves as the base for the engine's software. Intel-based PCs serve as Windmill's cost-effective hardware platform. Currently, Windmill is being used with broadcast or ring-based datalink layers, including Ethernet and FDDI; however a future implementation of the

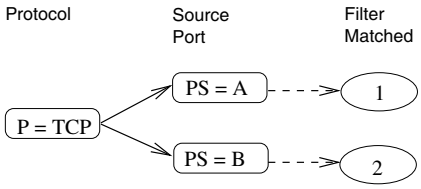

Figure 2. Simple DAG representation of two packet subscriptions.

architecture could utilize custom hardware splitting techniques to measure protocols over serial media in a manner similar to the OC3MON tool [20].

WINDMILL PROTOCOL FILTER

The probe's WPF passively examines all the underlying network traffic and performs *one-to-many* packet multiplexing to the probe experiments. It does this by constructing an intermediate representation of the outstanding subscriptions in the form of a directed-acyclic graph (DAG). This graph is dynamically compiled to a native machine language module, and is finally installed in the probe machine's kernel. For example, a request for TCP traffic with source port A from experiment 1 and a request for TCP traffic on source port B from experiment 2 would result in the DAG shown in Figure 2. WPF differs from past packet filters [12–16], where network packets are passively matched to a specification and demultiplexed to a single endpoint, in that it identifies a set of destinations for a packet. By determining a set of end-points, WPF avoids the subtle problem inherent in one-to-one matching algorithms of client starvation from overlapping filters.

One-to-many matching is motivated by the fact that a probe machine may be executing numerous concurrent experiments that are interested in some of the same packet streams. As the streams of packets arrive, the filter for each experiment must be used to determine which packets are sent to which experiments. This can be done at reception time, where each packet is compared to different experiments' filters. This would be similar to using multiple BPF (Berkeley Packet Filter) devices to do the determination, one for each experiment. Packets can also be matched to experiments by determining a packet's destinations before its reception. WPF adopts the latter approach in that it precomputes all possible combinations of overlapping filters when they are made, and generates a DAG to reflect these comparisons. Once the DAG is constructed, it is compiled to native machine language on-the-fly and installed in the kernel for matching against incoming packets.

Logically, a message header consists of a set of *comparison fields*. A filter is composed of a conjunction of predicates. Each predicate specifies a Boolean comparison for a particular field. An experiment registers a filter by supplying a set of values for one or more of these comparison fields. These fields can correspond to Internet protocol specific values, e.g. IP source address or TCP destination port, or they can be subsets of these values. This allows filtering based on fields such as the first 24 bits of the IP source address, commonly used to examine packets from a specific network.

Table I. Three overlapping packet filters and a sample input packet.

	Bit comparison elements				
Experiment	IP Src Addr	IP Dst Addr	Protocol	Src Port	Dst Port
Filter 1	AS = X	*	P = T	PS = A	*
Filter 2	*	AD = Y	P = T	*	PD = B
Filter 3	*	AD = Z	P = T	*	PD = C
Packet to match	AS = X	AD = Y	P = T	PS = A	PD = B

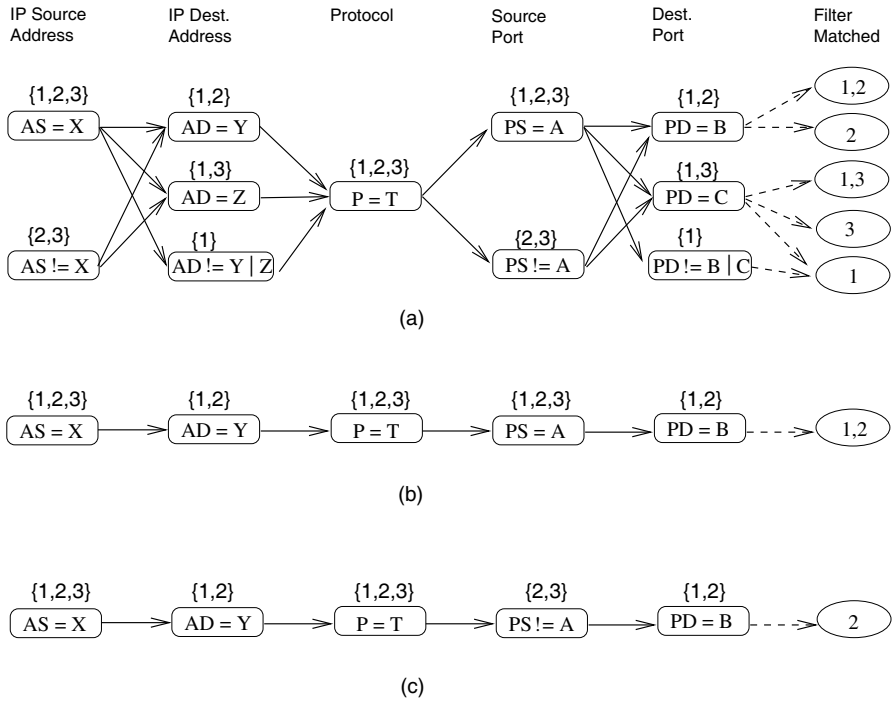

Figure 3. (a) The DAG generated from the three filters shown in Table I. (b) An example of a packet matching filters 1 and 2. (c) Follows a path through the DAG of a packet that only matches filter 2.

To illustrate how WPF works, consider the set of filters shown in Table I, with each filter representing the packet subscription from one experiment. The table shows five comparison fields as the basis for three experiments. Each entry in the table specifies the value to be matched, or a '*' representing a wildcard. For example, the sample input packet above matches both filters 1 and 2. The intermediate representation of these filters as a DAG is shown in Figure 3(a). The vertices represent Boolean operations on the comparison fields; a match results in a transition to the right. Furthermore, each vertex is also labeled with the set of corresponding filters when the Boolean operation associated with the vertex is true. Consider the vertex AS = X, which is labeled with the set {1, 2, 3}. This indicates that if the IP source address (AS) in the input packet is X, then the input packet matches all three filters for the field in question. Consequently, each path through the DAG corresponds to matching the input packet with a unique set of the filters. For example, an input packet that matches the path in Figure 3(b) satisfies both filters 1 and 2, but not filter 3. Similarly, an input packet that matches the path in Figure 3(c) also matches filter 2, but not filters 1 and 3. Observe that the intersection of the set of labels associated with the vertices on a path identifies the unique set of filters that match input packets.

To illustrate the subtle problem associated with packet filters that utilize most specific matching, reconsider the example in Table I. Note that none of the three filters is more specific than the others. As stated before, the sample input packet above matches both filters 1 and 2. In both PathFinder and DPF, the packet filter will supply the packet to the experiment that matches first in the corresponding trie data structure [15,16]. This can lead to starvation of packet destinations whose filter is not the first to match an incoming element. To correctly accommodate one-to-many matching using DPF or PathFinder, one would need to use as many trie structures as experiments, resulting in $O(mn)$ time complexity, where m is the number of experiments, and n is the number of comparison fields.

Once the subscriptions from each experiment have been combined into a master filter, the master filter is compiled into a single block of machine code and loaded into the running kernel. The goal of this compilation is to reduce the time needed to process each incoming packet. Depending on the type of requests we expect to receive from the experiments, there are several alternative methods for performing this compilation. We discuss these alternatives later when we present the details of the Windmill packet filter. The current version of the WPF assumes that the number of experiments is small enough for us to sacrifice space in the resulting code in exchange for speed. This results in filters that run as fast as other compiled filters.

ABSTRACT PROTOCOL MODULES

The abstract protocol modules export interfaces to probe experiments for both protocol reconstruction and the direct access to any protocol layer's events and data structures. Through these interfaces, the abstract protocol modules provide for the breadth and depth of protocol analysis as well as inter-protocol event correlation. Typically, network protocol layers are designed to hide the details of their underlying layers, and to provide some type of data frame or byte stream service to the layers above them. The abstract protocol modules are similar in that they can be chained together to build the service of higher layer protocols from the bare stream of data packets arriving from the WPF. For example, an experiment can read a TCP session's byte stream in either direction by supplying the TCP module with captured packets. However, the abstract protocol modules intentionally violate the encapsulation and abstraction of the lower protocol layers by exporting the details of these layers, including protocol

events and data structures. The probe experiments can then correlate these normally hidden data with the performance of higher layer protocols.

Each module exports its protocol abstraction through its interface. By chaining invocations to the modules, an experiment can infer the target protocol's behavior and performance at an end-host. For example, an experiment could monitor and measure the sequence of HTTP 1.1 requests over a persistent connection by supplying the HTTP module with the incoming data packets and making non-blocking `get_next_request` calls to the module. Moreover, the performance of the lower layer protocols are also accessible to the experiment through the module's interfaces. Extending the previous example, an experiment could measure the number of duplicate TCP acknowledgments in the underlying stream to infer the congestion along the HTTP connection's path. The current implementation of the Internet base protocol modules—IP, UDP, and TCP—borrows heavily from the 4.4BSD-Lite distribution's networking code [21] to determine protocol error conditions and functionality. New protocol modules can utilize the base protocol modules to implement their functionality with a minimum of effort. This effort would be proportional to the complexity of the original protocol's implementation. For example, implementing the IP protocol module took several hours; whereas the TCP module took weeks of debugging and refinement. A measurement-focused version of BGP took less than a day.

The abstract protocol modules are designed to minimize the amount of duplicated effort by a set of concurrently executing experiments. Just as the WPF makes one comparison for each field regardless of the number of experiments, the abstract protocol modules only recursively execute a protocol stack on an incoming packet once. This is achieved by explicitly managing and coordinating the packet processing in each protocol module. Packets are treated as objects which are managed by the abstract protocol modules' memory management library. The packet dispatch routine initializes a reference count on the incoming packets, which is explicitly decremented by the experiments when their data are no longer needed. Each packet object has references to protocol-specific data objects that are managed by their respective protocol modules. In addition to per-packet memory state, each module that supports a stateful protocol layer, such as TCP or HTTP, manages the memory required to keep its subscribed endpoints' state.

EXTENSIBLE EXPERIMENT ENGINE

While the abstract protocol modules are designed to provide experiments a unified interface into protocol reconstruction, the experiment engine is designed to support changing experiments over time. As the passive component of an Internet measurement infrastructure, Windmill must be able to dynamically load experiments, modify them during execution, and remove them when they are no longer needed. These features enable experiments to adapt to the needs of a larger measurement infrastructure, not only by responding to events at other measurement points but also by responding to the changing interests of the experimenters. For example, a large routing protocol study can continue running while a network administrator tries to monitor an active attack on the network.

Our current solution is to use the standard dynamic linker to link experiments as they are downloaded into the tool. To support dynamic linking, each experiment is first compiled into a shared library. From there, loading the experiment simply requires notifying the experiment engine of the

new experiment. Windmill also supports dynamic loading of the kernel filters through the loadable kernel module support in FreeBSD.

The current implementation of Windmill does not employ safety checks between loaded experiment modules. It is possible for an erroneous or malicious experiment to disable the probe. The use of an operating system or language safety devices could be employed to partition the experiments' data spaces from each other. However, functionality typically provided by real-time operating systems would be needed to fully partition the experiments from each other in terms of bounding computational resources.

The experiment engine exports an administration interface that allows for the remote management of the passive experiments. Currently, experimental results are either stored locally on disk for later retrieval through the interface, known as *measure-and-fetch*, or sent to a remote destination in real-time using a custom data dissemination service [22]. Using the data retrieval and dissemination service, the experimental results from many probes can be correlated to provide an aggregate performance profile of a target protocol over a given network topology. There are several projects investigating the aggregation of Internet performance data [1–3]. This difficult problem is outside the scope of this paper.

EXAMPLE APPLICATIONS

This section provides the results from two sets of experiments obtained using the current implementation of Windmill. The first experiment reconstructs the BGP interdomain routing protocol [23] to validate a possible correlation between Internet routing instability and network bandwidth usage. The second set of experiments demonstrates the tool's ability to measure a real-world system. It also shows how the probe can be used to reduce application dataflows to a manageable size through on-line data reduction. In addition it illustrates the ability of the passive probe to drive an active measurement infrastructure.

BGP experiments

In these experiments, a module was created to monitor and measure the BGP [23] routing traffic exchanged between two peer border routers in order to validate one of the key observations presented in [5]. Specifically, the authors found a strong correlation between Internet routing instability and network bandwidth usage. One possible reason for such a correlation was the use of TCP as the underlying transmission mechanism in BGP. This experiment evaluates the effects of network congestion on a router's stream of BGP messages exchanged between peers and presents evidence that suggests that BGP is modified for use with a UDP keepalive protocol.

BGP is an incremental routing protocol, in that upon connection establishment, two routers or peers that have agreed to exchange reachability information share their full routing tables with one another. Unlike many other routing protocols, this full exchange happens exactly once. After the initial exchange, a peer only shares routing information that varies—when routes to a destination prefix change. If there is no fluctuation in routes for some period of time, a keepalive message is sent to the peer. If a keepalive message or routing update is not received within a bounded time period, in this case the router's Hold Timer, the peering connection is severed causing the withdrawal of all the peer's routes—making them unreachable throughout the autonomous system and its downstream networks.

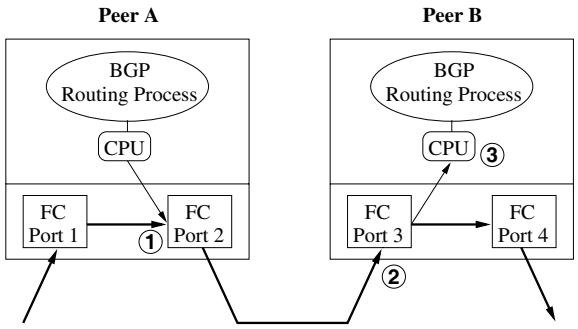

Figure 4. BGP experimental apparatus.

Subsequently, the connection is reestablished, resulting in a full exchange of the routing tables. This is seen by Internet endpoints as routing instability—the fluctuation of routing state affecting packet forwarding.

There are both positive and negative consequences of using TCP as BGP's underlying transport protocol [24]. The benefit of using TCP is the support for incremental routing through TCP's reliable byte-stream service. No routing data are lost between peers, making full routing table refresh messages unnecessary. The problem with using TCP is its adaptation to network congestion. When network congestion is at its worst, the peering session over which routing information is shared receives its least bandwidth. One would like the Internet infrastructure to stand rigid under maximal stress; however routing information and therefore infrastructure stability are hampered during high network loads. The hypothesis presented in [5] is that during peak network usage, the TCP session supporting a peer's stream of BGP update and keepalive messages is backed off due to congestive loss. This allows one of the peer's Hold Timers to expire before the message exchange between peering sessions completes. This would cause peering sessions to fail at precisely those times when the network load was greatest.

Figure 4 shows three possible places where a BGP session's TCP packets could be lost due to network congestion at an autonomous system (AS) boundary. To achieve high throughput, most production routers perform next-hop IP forwarding directly in the networking hardware. Therefore the forwarding cards (FCs), which handle the network devices, bypass the router's CPU for all traffic except that relating to Internet control. The router's BGP sessions fall in the latter category and are handled by its BGP process. This process runs on the router's operating system which is also managed by the CPU. In Figure 4, the overloaded route starts with IP packets coming into Peer A through the forwarding card handling Port 1. These packets are directly placed in the transmission buffer of the Port 2 forwarding card for transmission to Peer B through Port 3. There are three places outlined in the figure where an IP datagram from Peer A's BGP TCP session could get lost. First, if Port 2 is forwarding data from Port 1 (or the sum of any input ports directing data towards this destination) at its capacity, it is clear that the IP datagram could be lost when Peer A's operating system attempts to add the packet to the overflowing transmission buffer on Port 2's forwarding card. In the second case, it is possible that Port 3's incoming buffer is filled to capacity and drops the datagram upon reception.

Third and finally, it is possible that Peer B's operating system buffers are full due to some external factor, possibly handling the millions of duplicate withdrawals seen from a different BGP peer [5].

To test the hypothesis that network congestion was affecting the BGP TCP stream, we performed an experiment that mimics the setup described in Figure 4 on a testbed consisting of five 200 MHz Intel Pentium-Pro machines. All five were running FreeBSD 2.2.5 with 32 MB of RAM, and each had three 100 Mbps Ethernet adapters—Intel Ether Express PRO/100B PCI. Two of the machines, corresponding to Peer A and Peer B, were configured as routers running the Multithreaded Routing Toolkit's [25] implementation of BGP. The peering sessions' Hold Timers were set to the default 180 s, resulting in the generation of periodic keepalive messages at 30 s intervals—the only BGP messages exchanged on the otherwise idle peering session. Two machines were used as input drivers to Peer A that blasted IP datagrams through two point-to-point Ethernet connections. These datagrams were routed in Peer A's kernel to the Ethernet connecting Peer A and Peer B. The fifth machine, running the Windmill BGP experiment, subscribed to the connecting Ethernet's BGP traffic on TCP source or destination ports 179 and reconstructed the peering sessions through the use of the TCP abstract protocol module. We were able to cause the BGP peering session to fail every time we ran the experiment.

The hypothesis that the BGP experiment validates is that network congestion at the router causes the peering session's TCP to back off, and prematurely severs the connection. Windmill provided several mechanisms that allowed the construction of a simple experiment module to validate this conjecture. These include support for answering the following questions.

- Was a peering session closed prematurely? Was there any unacknowledged data observed in the connection's pipe?
- Is there evidence of congestion along the path? Are there any missing sequences of data in the data window?
- Are the gaps due to an observed error in the datapath?
- Is the peering termination consistent with the advertised BGP hold timer? What is the difference between the time of the last acknowledged keepalive and the FIN packet?

The Windmill's control flow is handled using events. The most common events are incoming packets that match an experiment's flow subscriptions. These events are presented to an experiment through a callback routine provided by the experiment. The experiment passes the packets to the appropriate abstract protocol modules, which reconstruct protocol state and manage the statistics for their respective layers. Upon return, various statistics that are triggered by the packet—aggregate events—can be identified by the experiment and handled. In addition to packet events, the Windmill provides support for time-based events.

The Windmill's TCP abstract protocol module reconstructs the byte-stream service and maintains statistics for both directions of a TCP session. There are experiment specific data structures that are associated with each direction of a TCP data stream. References to these data are managed by the TCP module. There are many actions the TCP module can take on behalf of an experiment when processing an incoming packet. For some applications it may be unnecessary for the TCP module to reconstruct the byte-stream service, where only a verification of TCP header and length fields is necessary. An experiment explicitly directs the TCP module to keep desired statistics using the `tcpEnableStats` call. One of the basic TCP statistics to be kept is the byte-stream service and its accompanying Data Window. We use the term data window to denote any unacknowledged data that have been observed by Windmill.

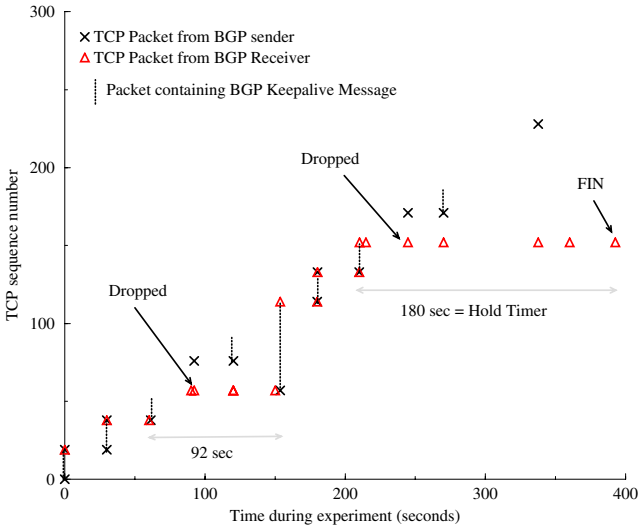

Figure 5. The collapse of a BGP peering session, due to multiple IP datagram drops in the congested TCP stream. Both peers are generating keepalive messages at 30 s intervals; the sequence numbers correspond to the TCP stream originating at the overloaded peer.

Upon the collapse of a peering session, the Windmill's BGP experiment wrote various statistics to disk, including a bounded window of packet contents. One such collapse is graphically represented by Figure 5. This figure shows that the IP datagrams containing the TCP segments transporting the BGP keepalive messages sent from Peer A to Peer B are dropped at Peer A's Port 2 device driver due to a full transmission buffer. The congested TCP connection induces Peer B's Hold Timer to expire, causing the BGP peering session to collapse. The crosses denote TCP packets sent by Peer A that are observed by the experiment, whereas the triangles represent those sent by Peer B. The short vertical dotted lines mark Peer A's TCP packets that fully contained a BGP keepalive message (19 bytes in length)—the only data exchanged over the TCP connection. Both the first three keepalives and their acknowledgment are measured by the experiment, however the datagram that contained the fourth keepalive is dropped. At time 120 the datagram containing the fifth keepalive is observed, which leaves a gap in the TCP byte-streams. This event causes the receiver to reacknowledge the receipt of the third keepalive's sequence number. After 92 s, a datagram containing the fourth, fifth, and sixth keepalives is received, thus resetting the Hold Timer before expiration. However at time 240, the ninth keepalive's datagram is dropped. This time the protocol never recovers. The following keepalive's datagram is received, however this causes the receiver to ask for a retransmission. This retransmission is lost, along with any other data. Note that at time 340, an acknowledgment packet is observed from Peer A with a sequence number 228, indicating that datagrams containing the 11th and 12th keepalives were introduced into the pipe but dropped. Finally after 180 s the experiment observes a TCP datagram with the FIN bit set from the Peer B, resulting in the termination of the peering session.

We observe that a BSD kernel routes packets with a fastpath similar to that used by hardware forwarding cards. Peer A's kernel will forward the overloading incoming datagrams directly into the transmission buffer of the outgoing port's device driver without user-level intervention. This simulates the direct transfer that occurs with the hardware forwarding cards. Peer A's outgoing TCP session's datagrams are dropped when the kernel attempts to add them to an already overfull transmission buffer.

While this result applies to BGP peering sessions at exchange points, it is perhaps more significant when applied to IBGP speakers—those BGP peering sessions within an AS. IBGP peering sessions traverse interior routers in an autonomous system to fully connect all of the system's border routers. Through these connections, the border routers agree on the routes exported and policies applied to their neighboring autonomous systems. The probability that an IBGP session's datagrams are lost at one of these congested interior routers is possible under heavy network loads, making this result even broader.

We believe that by using a combination of both UDP heartbeats and TCP routing exchanges, a BGP implementation would be less affected by network congestion. Had BGP used a UDP-based heartbeat in these experiments, the peering session would have been significantly more robust in the face of congestion. An alternative solution is for routers to mark the BGP flows with an appropriate Type of Service, which are then given a higher priority. Some routing vendors have begun to implement this; however it is unclear whether the traffic generated on the router itself is affected. These solutions both increase the stability of BGP peering sessions and would directly strengthen the Internet infrastructure.

Collaboratory experiments

This set of experiments demonstrates the use of Windmill in a real-world setting by instrumenting a key server in the Upper Atmospheric Research Collaboratory (UARC) to gather a broad range of statistics. The use of Windmill for on-line data reduction is illustrated by the collection of application-level statistics. Specifically, we extract statistics from the application's data flows that could not be done using post analysis due to the volume of data. Moreover, the experiments show how Windmill can be used in conjunction with an active measurement infrastructure to obtain snapshots of network metrics that can be temporally correlated with passive statistics. Finally, all of these statistics were gathered without modifying the UARC software or host operation systems; as such it represents an example of utilizing passive techniques for measuring *shrink-wrapped* systems.

UARC is an Internet-based scientific collaboratory [26]. It provides an environment in which a community of space scientists geographically dispersed throughout the world perform real-time experiments at remote facilities. Essentially, the UARC project enables this group to conduct team science without ever leaving their home institutions. This community has extensively used the UARC system for over four years. During the winter months, a UARC campaign—the scientists use the term *campaign* to denote one of their typically week-long experiments—occurs around the clock. The UARC system relies on a custom data distribution service [22] to provide access to both real-time and archived scientific data. In these experiments Windmill was deployed to measure one of the system's central data servers during the April 1998 scientific campaign. In order to provide ubiquitous access to the UARC system, users access the system through the Web via a Java applet. One consequence of this decision was the implementation of the data distribution as multiple TCP streams between UARC servers and the client browsers. During this experiment, Windmill intercepted all of the data communications between the main UARC server and its clients. By reconstructing the TCP and application-level sessions from these flows, Windmill extracted the UARC data.

The UARC system provides access to data from over 40 different instruments from around and above the world including the ACE, POLAR, JPL GPS, and WIND satellites; Incoherent Scatter Radar arrays in Greenland, Norway, Puerto Rico, Peru, and Massachusetts; magnetometers; riometers; digisondes; and real-time supercomputer models. These instruments supply over 170 distinct data streams to the scientists. The goal of our experiments was to obtain user-level performance statistics for analysis by behavioral scientists, such as when and to which instruments the users connected, and what time ranges of data they requested. These statistics can be correlated with chat room logs to model collaboration at a very high level. Similarly, we wanted to determine what effect a user's network connectivity had on her participation. A full analysis of these experiments is outside the scope of this paper; this section focuses on how Windmill made these measurements possible and only summarizes the findings.

The passive measurement of any serial connection requires hardware intervention. The UARC server was originally connected to the Internet through a 100 Mbps switched Ethernet port on a Cisco 5500 router. For these experiments, we split the switched Ethernet by inserting an Intel Express 10/100 Stackable Hub between the router and the server. The perturbation of the system was the addition of an extremely small amount of latency. Windmill ran on a 300 MHz Pentium-II based PC with 128 MB RAM.

The use of Windmill for on-line data reduction is illustrated by the collection of user-level statistics for the UARC behavioral scientists. These statistics correspond to actions initiated by the users, including addition and removal of subscriptions to data suppliers, as well as requests for archived data. In order to measure these application-level statistics, the UARC transport protocol was reconstructed. Like BGP, its frames are built on top of TCP and it uses a fixed header size with variable size data payloads. Only a small fraction of the application-level frames exchanged between the client and server describe user actions; the majority of the traffic is scientific data. Three days of continuous campaign throughput was reduced by over five orders of magnitude to approximately 200 KB of statistics. The following is a subset of the questions that these measurements answered.

- *Determination of the amount and level of synchronous collaboration.* What are the duration and times when the scientists' view of the data overlapped, enabling concurrent analysis? This corresponds to determining when the scientists were in the same virtual room at the same time.
- *Investigation into amount of cross specialization activity.* Do the scientists focus only on the instrument and supply types that define their specialty, or do they exploit the wealth of data made available by the system?
- *Temporal access patterns of the scientists.* An analysis was done to determine whether the scientists have changed their access to their data. In the past, when they were colocated, they would all sit in a quonset hut and engage in science. Does this continue with a dispersed community?
- *Access patterns to archived data.* How does the ability to access over a year's worth of archived scientific data impact their real-time campaign? The experiment measured the access patterns to the archived data as well as the real-time supplies.

Windmill's passive measurement and analysis of underlying data streams is extremely powerful when coupled with active measurements. Windmill allows experimenters to define *trigger events* that can be used to initiate active measurements in an external tool. For example, when a flow's bandwidth drops below a threshold, an experiment could generate a trigger event that instructs an

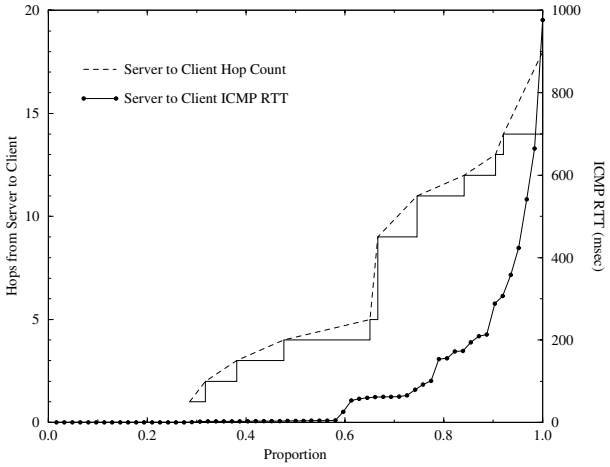

Figure 6. Distributions of initial server to client hops and ICMP round trip times. Hop-count is a cumulative distribution; ICMP RTT is a sorted distribution.

external tool to obtain a path snapshot through a mechanism such as `traceroute`. Typically, if using passive measurement techniques for off-line analysis, it is not possible at analysis time to gather additional data about the state of the network due to the transient characteristics of Internet paths [27]. However, by pairing passive with active measurements, a broader range of statistics can be obtained. To illustrate this feature, in conjunction with the UARC data reduction experiment, Windmill ran an experiment that classified the different types of client–server connections. Specifically, when the experiment recognized a connection from a previously unconnected host, it sent a message to an active measurement probe running on the same machine that performed a path and ICMP RTT measurements on the client. Figure 6 shows the results from this initial active measurement of newly connected client hosts. Both lines represent a proportion. Note that for ease of comparison, the normally independent horizontal axis has been made dependent. For the server to client hop-count, the horizontal length of the stair-step represents the following data point's proportion. For example, roughly 70% of the client hosts were five hops away from the UARC server. The ICMP round trip times are shown not as a cumulative distribution, but instead represent the measured RTT for their respective hosts.

During a connection's lifetime, its effective bandwidth was measured by the experiment. When the bandwidth fell below a threshold, Windmill sent another message to the active probe, which then obtained another set of `traceroute` and `ping` statistics. In this way, a profile of the client–server connections were collected and correlated with the higher level behavioral data. There is an inherent bias in passively estimating a TCP session's bandwidth that depends on both the manner in which the estimate is taken, and the distance of the probe from either of the endpoints. For this experiment, bandwidth was estimated using the following formula:

$$B_{\text{est}} = \frac{N - M}{T_{\text{ack}\,N} - T_{\text{ack}\,M}}$$

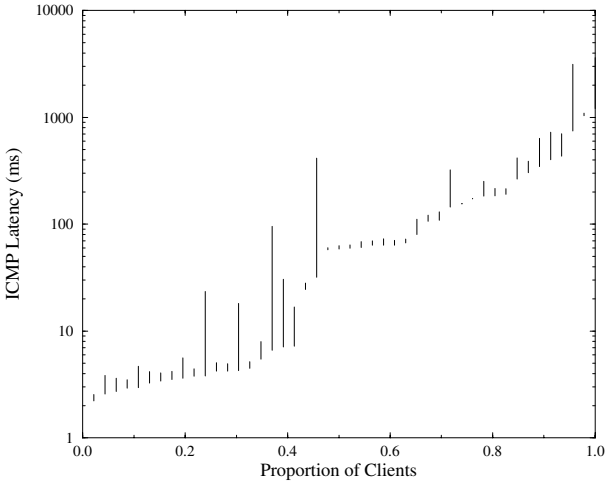

Figure 7. Server to client ICMP round trip means and variations.

T_{ack} represents the measured times at which the first acknowledgments were seen for sequence numbers M and N. The bandwidth was only estimated when there was a difference of 15 KB between N and M. Since this estimate is made near the server, it is similar to the rough bandwidth estimate graphically presented to an FTP user after performing a `PUT` command. Over the course of the campaign, the experiment only observed four routes that changed. Three of these four routes were 11 hops or greater in length, and one was a 5 hop dial-in route that had intermittent connectivity. The experiments measured more variance in ICMP round trip times than in path length. Figure 7 shows these results. In this figure, the lower point on a vertical line represents the mean ICMP round trip time for a single client; the upper point on the vertical line denotes the mean plus a standard deviation. The clients are evenly distributed across the horizontal axes, sorted by increasing mean latency. The round trip variation results show a two-tiered connectivity demographic. The lower 40% of the clients are well connected to the UARC server through service providers located in the continental United States. The remainder of the participants are located elsewhere.

WINDMILL PACKET FILTER DETAILS

Having discussed the basic architecture of Windmill as well as several examples of its use, this section delves into the details of the Windmill Packet Filter implementation and performance. Given our goal of providing one-to-many multiplexing of incoming packets to multiple concurrent experiments, we needed to devise a way of compiling our master filter into fast assembly code. We discuss several design alternatives and then discuss the details and performance results from the design we chose to implement.

Given the WPF's intermediate representation of a set of filters as a DAG, our goal is to reduce the time complexity of matching an input packet with a set of filters. First, we need to define a few terms. Suppose that we have specified m filters on n comparison fields. A set of filters is defined to be *sparse* if the number of distinct values for each (comparison) field is a small constant, i.e. much smaller than m, the total number of filters. A set of filters is defined to be *dense* if the number of distinct values for at least one of the (comparison) fields is $\Theta(m)$. Note that by definition these properties are mutually exclusive. If a set of filters has one field with $\Theta(m)$ distinct values (dense), it will have at least one field with more than a small constant number of distinct values (not sparse). We propose three complementary optimization techniques:

Alternative 1—Path enumeration for sparse filters. By enumerating all possible paths in a WPF DAG a priori, one can perform a match on a given input packet in $O(n)$ time. As in DPF and PathFinder, a hash table is used at points where there is more than one value for a comparison field. However, this improvement in time complexity does not come for free. We have effectively traded time complexity for space complexity since there is potentially an exponential number of paths in a WPF DAG. However, if the set of filters is sparse, then the maximum branching factor for each vertex is effectively a very small constant. This property significantly reduces the number of paths in a DAG.

Alternative 2—Set intersection for dense filters. This alternative does not require enumeration of all possible paths in a WPF DAG. It exploits a subtle property of dense filters to obtain a running time complexity of $n \log(m)$ for matching a packet to a set of filters without additional space overhead. If the set of filters is dense, then there is a comparison field such that the total number of distinct values for that field is roughly m. By switching this field to the front of the WPF DAG, one can determine the intersection of a pair of labels in $\log(m)$ as a path is traversed in a WPF DAG. Note that one label has a constant number of items whereas the other is an ordered set of at most m items.

Alternative 3—Bit mask representation. Finally, we observe that a common case may involve the deployment of probe machines running no more than several hundred concurrent experiments. In such scenarios, one can represent the label associated with each vertex as a bit mask. The cost of obtaining the intersection of the labels is reduced to the cost of performing an AND operation on the labels. The cost of matching an input packet grows slightly as the number of concurrent filters increases.

The current design of the WPF assumes that we are using a small number of filters, at most 32, and that we are much more interested in saving time than space. For that reason we use the first alternative discussed above to generate filters. Each possible path in the DAG is enumerated to create a new DAG that we represent as a finite state machine (FSM). Computer science has long used the notion of a finite state machine to represent a pattern recognizer [28]. This representation provides a simple model for merging separate machines and can be easily modified to handle returning more than a simple Boolean. In addition, discrete event systems use finite state machines as a fundamental model for developing control systems. Discrete event systems use composition of FSMs to simplify the construction of a complex system; an operation that is very similar to the process we use to combine several filters into one master filter [29]. Another advantage of the FSM representation is the notion that all the state for a

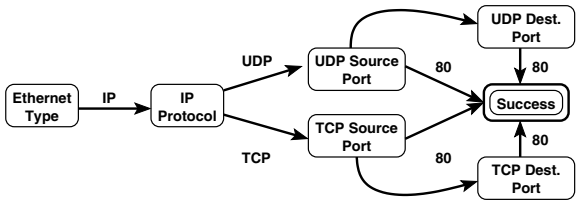

Figure 8. Sample finite state machine.

partial match is represented solely by the state the machine is currently in. This allows us to combine redundant states when merging in order to save space. Finally, the FSM model translates easily into machine code, greatly simplifying the compiler design.

An example of a filter that represents the `tcpdump` string 'port 80' is shown in Figure 8. Each state of the FSM represents a specific field to be matched. Each edge represents a value for the preceding field, with non-labeled edges representing failure transitions. The failure transitions are followed if none of the other values match a given field. Final states, labeled 'Success' in this example, are normally marked with the set of matching states. When the filter is run against an incoming packet, the set of matching experiments is passed to the packet dispatcher along with the packet.

CONCLUSIONS

The architecture and implementation of Windmill, a passive network protocol performance tool, have been presented. Windmill enables experimenters to measure a broad range of protocol performance metrics by both reconstructing application-level network protocols and exposing the underlying protocol layers' events. This range encompasses low-level IP, UDP and TCP events such as packet corruption and length errors, duplications, drops, and reorderings, as well as application-level performance characteristics. By correlating these inter-protocol metrics, the normally hidden interactions between the layers are exposed for examination.

Windmill was specifically designed as the passive component of a larger Internet measurement infrastructure. This work complements the efforts of several research groups that are deploying active measurement probe infrastructures within the Internet [1–3]. Unlike most tools that focus on capturing data for post analysis, Windmill was designed to support continuous passive measurements at key network vantage points. The architecture allows application-level protocol data to be distilled at the measurement point for either on-line analysis or further post analysis. The extensible architecture enables experiment managers to vary the number and scope of Windmill's experiments.

The key contributions of our architecture are its fast WPF and the support for both experiment extensibility and high-performance protocol reconstruction. Through the combination of dynamic compilation and a fast matching algorithm, Windmill's WPF can match an incoming packet with five components in less than 350 ns. Additionally, WPF addresses some fundamental limitations in past packet filtering technology by correctly handling overlapping filters. Windmill enables the

dynamic placement, management, and removal of long-running experiments, while accommodating the significant demands for protocol reconstruction performance.

In order for the rational growth of the Internet to continue, a deeper understanding of the interactions between its protocols is needed. As an implementation of a passive application-level protocol performance measurement device, Windmill can be used to explore these interactions in real-world settings. As an example of this use, the paper presented results from a BGP experiment that identified a possible cause for the correlation between routing instability in the Internet [5] and high levels of network congestion. Specifically, the use of TCP as the underlying transport mechanism for BGP peering session keepalive messages was shown to collapse under high levels of congestion. The experiment suggests that BGP be modified for use with a UDP keepalive protocol. These experiments illustrate the ability of Windmill to correlate lower-layer protocol events, namely dropped TCP segments, with the high-level BGP protocol transmission of keepalive messages.

This paper also presented a set of experiments that measured a broad range of statistics of an Internet Collaboratory. These experiments highlighted the ability of Windmill to perform on-line data reduction by extracting application-level performance statistics, such as measuring user's actions, and demonstrated the use of the passive probe to drive a complementary active measurement infrastructure. These experiments highlight the ability of Windmill to measure Internet infrastructure, such as servers, without modifying end-host application or operating system code.

We believe that passive measurement techniques will become increasingly important as the commercial shift in the Internet continues. The ability to measure *shrink-wrapped* protocol implementations is critical due to the overwhelming deployment of commercially based protocol implementations in both the Internet's end-host and infrastructure nodes. Together, the inability to take the system off-line or modify it for study implies the need for increased passive measurement of Internet performance. Windmill was developed for precisely this purpose.

ACKNOWLEDGEMENTS

This work was supported in part by grants by the National Science Foundation (NCR-9710176) and by DARPA grant number F001670. The Intel Corporation made the BGP experiments possible through its significant donation of lab equipment through the Education 2000 program. Rob Malan was supported by an IBM Graduate Fellowship during part of this work. Sugih Jamin and Peter Knoop provided helpful feedback during the conceptual design. Insightful comments from Vern Paxson and Gary Delp significantly improved the conference version of this paper.

REFERENCES

1. Mahdavi J, Mathis M, Paxson V. Creating a national measurement infrastructure (NIMI). *Internet Statistics and Metrics Analysis (ISMA) '97*, May 1997.
2. Almes G. Metrics and infrastructure for IP performance. http://www.advanced.org/surveyor/ [September 1997].
3. Internet Performance Measurement and Analysis (IPMA) project homepage. http://www.merit.edu/ipma/ [May 2003].
4. IP Performance Metrics (IPPM). http://www.ietf.org/html.charters/ippm-charter.html [May 2003].
5. Labovitz C, Malan R, Jahanian F. Internet routing instability. *Proceedings of ACM SIGCOMM '97*, Cannes, France, September 1997. ACM Press, 1997.
6. Zhang Y, Breslau L, Paxson V, Shenker S. On the characteristics and origins of internet flow rates. *Proceedings of ACM SIGCOMM*, August 2002. ACM Press: Pittsburgh, PA, 2002.
7. Feldmann A. BLT: Bi-layer tracing of HTTP and TCP/IP. *WWW-9/Computer Networks*, vol. 33. Elsevier: Amsterdam, 2000; 321–335.

8. Shannon C, Moore D, Claffy KC. Beyond folklore: Observations on fragmented traffic. *IEEE/ACM Transactions on Networking (TON)* 2002; **10**(6):709–720.
9. tcpdump. http://www.tcpdump.org [May 2003].
10. Keys K, Moore D, Koga R, Lagache E, Tesch M, Claffy K. The architecture of Coral Reef: An Internet traffic monitoring software suite. *PAM 2001 Workshop*, April 2001. Ripe NCC: Amsterdam, 2001.
11. Anagnostakis KG, Ioannidis S, Miltchev S, Ioannidis J, Greenwald M, Smith JM. Efficient packet monitoring for network management. *Proceedings 8th IEEE Network Operations and Management Symposium (NOMS)*, Florence, Italy, April 2002. IEEE Communications Society, 2002.
12. Mogul JC, Rashid RF, Accetta MJ. The packet filter: An efficient mechanism for user-level network code. *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, November 1987. ACM Press: Austin, TX, 1987; 39–51.
13. McCanne S, Jacobson. V. The BSD packet filter: A new architecture for user-level packet capture. *Proceedings of the 1993 Winter USENIX Technical Conference*, San Diego, January 1993. Usenix Association: San Diego, CA, 1993.
14. Yuhara M, Bershad B, Maeda C, Moss E. Efficient packet demultiplexing for multiple endpoints and large messages. *Proceedings of the 1994 Winter USENIX Technical Conference*, January 1994. Usenix Association: San Francisco, CA, 1994.
15. Bailey ML, Gopal B, Pagels MA, Peterson LL, Sarkar P. PathFinder: A pattern-based packet classifier. *Proceedings of First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994. Usenix Association: Monterey, CA, 1994.
16. Engler D, Kaashoek MF. DPF: Fast, flexible message demultiplexing using dynamic code generation. *Proceedings of ACM SIGCOMM '96*, August 1996. ACM Press: Stanford, CA, 1996.
17. Anderson JM, Berc LM, Dean J, Ghemawat S, Henzinger MR, Leung S-TA, Sites RL, Vandevoorde MT, Waldspurger CA, Weihl WE. Continuous profiling: Where have all the cycles gone? *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997. ACM Press, 1997.
18. Zhang X, Wang Z, Gloy N, Chen JB, Smith MD. System support for automatic profiling and optimization. *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997. ACM Press, 1997; 15–26.
19. Rosenblum M, Herrod S, Witchel E, Gupta A. Complete computer system simulation: The SimOS approach. *IEEE Parallel and Distributed Technology* 1995; **3**(4):34–43 (Winter).
20. Apisdorf J, Claffy K, Thompson K, Wilder R. OC3MON: Flexible, affordable, high-performance statistics collection. *Proceedings of INET '97*, Kuala Lumpur, Malaysia, June 1997. Internet Society, 1997.
21. Wright GR, Stevens WR. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley: Reading, MA, 1995.
22. Malan GR, Jahanian F, Subramanian S. Attribute-based data dissemination for internet applications. *Journal of High Speed Networks: Special Issue on Multimedia Networking* 1998; **7**(3,4):319–337.
23. Lougheed K, Reckhter Y. A border gateway protocol (BGP). *RFC 1163*, June 1990.
24. Huitema C. *Routing in the Internet*, Prentice-Hall: Englewood Cliffs, NJ, 1995; ch. 8.2.
25. Multithreaded Routing Toolkit. http://www.merit.edu/mrt/ [May 2003].
26. Subramanian S, Malan GR, Shim HS, Lee JH, Jahanian F, Prakash A, Knoop P, Weymouth T, Hardin J. Software architecture for the UARC web-based collaboratory. *IEEE Internet Computing* 1999; **3**(2):46–54.
27. Paxson V. End-to-end routing behavior in the Internet. *Proceedings of ACM SIGCOMM '96*, August 1996. ACM Press: Stanford, CA, 1996.
28. Hopcroft JE, Ullman JD. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley: Reading, MA, 1979.
29. Cassandras CG, Lafortune S. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers: Dordrecht, 1999.

