An Extensible Probe Architecture for Network Protocol Performance Measurement

G. Robert Malan and Farnam Jahanian
Department of Electrical Engineering and Computer Science
University of Michigan
1301 Beal Avenue
Ann Arbor, Michigan 48109-2122
+1 734 647 8086
{rmalan, farnam}@eecs.umich.edu

1. Abstract

This paper describes the architecture and implementation of Windmill, a passive network protocol performance measurement tool. 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. 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, the results from several experiments are presented. The first set of experiments suggests a possible cause for the correlation between Internet routing instability and network utilization. The second set of experiments highlights: Windmill's ability to act as a driver for a complementary active Internet measurement apparatus, its ability to perform online data reduction, and the non-intrusive measurement of a closed system.

1.1 Keywords

Protocol performance, passive measurement, online analysis, packet filter

2. Introduction

The Internet has grown significantly, and shows little sign of stopping. 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 routers and highly trafficed 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 [25], Surveyor [1], and IPMA [12] projects. These projects have approached Internet measurement by utilizing active performance metrics [24] – 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, in that 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. Unlike most tools that focus on capturing data for post analysis, Windmill was designed to support 24x7 – 24 hours, 7 days a week, around the clock – passive measurements at key network vantage points. The architecture allows application-level protocol data (such as BGP, HTTP, DNS, etc.) to be distilled at the measurement point for either on-line analysis or further post analysis. Our tool allows for their measurement without modifying either the Internet infrastructure (the routers, nameservers, Web caches, end hosts, etc.) or host implementation of these protocols.

Windmill's architecture supports the passive performance measurement of application-level protocols through the use of protocol reconstruction and abstraction-breaching protocol event 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

This paper appears in the Proceedings of the ACM SIGCOMM '98 Conference.

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, round trip estimates – with the behavior and performance of the reconstructed application-level protocol.

To accommodate both performance and extensibility the probe'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 this work are:

- *Implementation of our passive probe architecture*: Our current implementation is built on 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.
- *Providing Experiment Extensibility*: The intrinsic trade-off between performance and extensibility was explicitly addressed in our architecture 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 run-time 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 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 demultiplexing 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 systems; and in practice WPF can match a five element filter in less than 350ns on a 200MHz Pentium-Pro. Moreover, WPF addresses a limitation of 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 [14]. Specifically, the experiment provides a possible answer for the correlation between Internet routing instability and network utilization. The experiment suggests that the BGP protocol be modified with a UDP keepalive protocol.
- *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.

The remainder of the paper is organized as follows. Section 3 places our work within the broader context of related work. Section 4 describes the probe architecture and current implementation. Section 5 presents the results from both the BGP and collaboratory experiments. Finally, Section 6 presents our conclusions and plans for future work.

3. Related Work

Passive techniques have been used in many low-level protocol performance evaluation and modeling studies (examples include [20, 5, 6]). 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 tool is targeted at the passive measurement of higher application-level protocol layers, and the correlation of lower layer protocol events with their performance.

One of the most widely used tools for Internet protocol monitoring is `tcpdump` [13]. `tcpdump` acquires network frames from an underlying filter and can either store those frames in binary or output the frame's IP protocols' header contents in ascii. However, `tcpdump` lacks the ability to reconstruct higher layer protocols from the underlying packet stream. In contrast, our tool provides the underlying filtering mechanism as well as the ability to reconstruct the high-level protocol streams.

The OC3MON tool captures low-level protocol headers over serial ATM network trunks for post analysis[3]. OC3MON is currently used for capturing IP, UDP, and TCP headers at points in the vBNS network.

There has been significant work done on packet filters. Traditionally, they have used interpreted code [19, 17, 28] or the traversal of high-level data structures [4] for safety and portability; however Engler's DPF [10] uses dynamic code generation techniques for high performance packet demultiplexing. One of our performance tool's components, the Windmill Protocol Filter (WPF), borrows from DPF in that it utilizes dynamic code generation for fast packet matching; however it differs from previous work in that it is designed to demultiplex packets to a set of receivers (one-to-many). In addition, WPF correctly demultiplexes packets to overlapping (ambiguous) filters.

Paxson's `tcpanaly` [23] is an offline tool for analyzing TCP traces. It classifies TCP implementations based on the characteristics seen in their traces. In order to classify a TCP connection `tcpanaly` must make two passes over the data stream. This is the only way certain characteristics can be identified. In contrast, Windmill is targeted at the analysis and distillation of application-level protocols at the measurement point in real-time for continuous network measurement.

There have been several fault-injection studies done on network protocols [8, 9]. Comer and Lin [8] injected faults into a TCP session from an end-point to probe for implementation errors. Similarly, Dawson [9] used an interposition agent on end-host systems for injecting faults into a protocol stream to measure protocol correctness. Through modification of an end host's protocol stack, both of these studies actively perturb the network traffic to validate protocol correctness. In contrast, the primary focus of our work is on performance measurement of network protocols through passive means from a neutral host. We show in Section 5.2 how the addition of an active measurement apparatus can be used to complement this passive technique.

Measuring distributed systems and their protocols at the end host can be done either 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: *shrink-wrapped* software, and production (real-world continuous-use) systems. A method for performing continuous system profiling with low overhead is presented in [2]. 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 [26]; however, this is not practical for measuring production systems where real-time events drive the protocols of interest.

4. 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 Windmill Protocol Filter (WPF) matches the underlying network traffic against a dynamically compiled filter. This filter is constructed from the set of outstanding packet subscriptions from Windmill's concurrent experiments. 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 execution of the protocol stack on incoming packets. The extensible experiment engine provides a mechanism for the loading, modification, and execution of probe experiments. Additionally, the experiment engine provides interfaces for the storage and dissemination of experimental results.

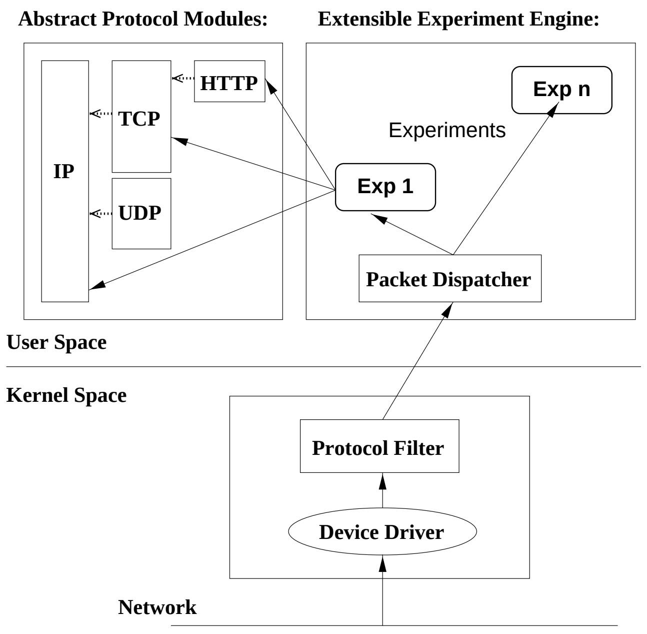

Figure 1. Organization of Windmill's architecture.

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 demultiplexing 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. The abstract protocol modules are invoked at the target protocol by the experiment which then recursively executes the lower layers of the protocol stack against the incoming packet. 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 a custom version of the FreeBSD 2.2.5 kernel and runs on an

Intel-based PC hardware platform. Currently, Windmill is being used with broadcast and ring-based datalink layers, specifically Ethernet and FDDI.

4.1 Windmill Protocol Filter

The Windmill Protocol Filter (WPF) passively examines all the underlying network traffic and performs *one-to-many* packet demultiplexing 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); dynamically compiling this graph to a native machine language module; and finally installing this module in the probe machine's kernel. WPF differs from past packet filters [10, 4, 28, 17, 19], 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. In this model, an experimenter can add or remove small experiments to Windmill as needed. 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 either at reception time, where each packet is compared to different experiments' filters (this would be similar to using multiple BPF devices to do the determination, one for each experiment); or 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 the subscriptions 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 sequence of predicates merged by conjunctions. 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 correspond to Internet protocol specific values (*e.g.*, IP source address, TCP destination port, etc.). In the current implementation, WPF is specifically targeted at Internet protocols, and has knowledge of UDP and TCP header lengths; as such is it not meant to be used as a generic packet filter.

To illustrate the subtle problem associated with packet filters that utilize most specific matching, consider the example in Table 1. The table shows five comparison fields as the basis for three experiments (with one filter per experiment). Note that none of the three filters is more specific than the others. For example, 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

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
Incoming Pkt:	AS=X	AD=Y	P=T	PS=A	PD=B

Table 1: Three overlapping packet filters and a sample input packet.

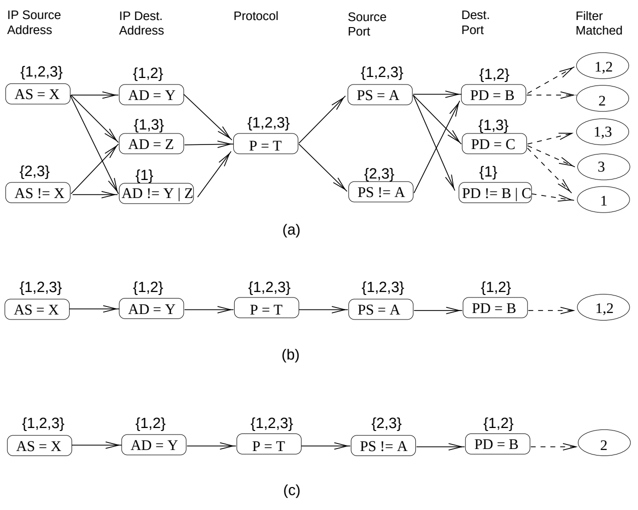

Figure 2. Part (a) represents the DAG generated from the three filters shown in Table 1. Part (b) shows an example of a packet matching filters 1 and 2. Part (c) follows a path through the DAG of a packet that only matches filter 2.

data structure [10, 4]. 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).

To illustrate how WPF works, consider the intermediate representation of the filters in this example as a DAG shown in Figure 2(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. For example, consider the vertex *AS=X*, which is labeled with the set {1,2,3}. It 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 subset of the filters. For example, an input packet that matches the path in Figure 2(b) satisfies both filters 1 and 2, but not filter 3. Similarly, an input packet that matches the path in Figure 2(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 subset of filters that match input packets.

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 terminologies. 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 *O(m)*. 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 are 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 are 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 WPF DAG. It exploits a subtle property of dense filters to obtain a running time complexity of *nlog(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)* time 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 as a path is traversed 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.

To give a concrete idea of WPF's performance, to match an unwound five element DAG – including IP source and destination, IP protocol, and TCP source and destination ports – takes less than 320ns on a 200Mhz Pentium-Pro (measured with the Pentium cycle counter).

4.2 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 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 only 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 this 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 `getNextRequest` 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 [27] to determine protocol error conditions and functionality.

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. For example, the TCP module keeps track of a stream's state so that reassembly only happens once, regardless of the number of subscribed experiments. 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 is 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 their subscribed endpoints' state. Section 5.1

provides a detailed example of a probe experiment's use of the TCP module.

The protocol modules also provide a namespace for packet subscription. Experiments indirectly subscribe to WPF packet streams by using the protocol modules to construct field comparison elements from layer-specific names. For example, an experiment could ask to subscribe to HTTP traffic by only giving the server's IP address to the HTTP module. This would generate several bit comparison elements with IP source and destination addresses, IP protocol set to TCP, and the TCP ports set to the default HTTP port.

4.3 Extensible Experiment Engine

As the passive component of an Internet measurement infrastructure, Windmill requires the ability to dynamically load experiments, modify them during execution, and remove them when no longer needed. These features enable experiments to adapt to the needs of a larger measurement infrastructure. Our current solution is to employ a custom runtime loader that can dynamically link experiments as they are downloaded into the tool. We had been pursuing a parallel line of development using a Java Virtual Machine with Just-In-Time (JIT) compilation, but performance considerations in moving to very high-speed monitoring have made it impractical.

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 (measure-and-fetch), or sent to a remote destination in real-time using a custom data dissemination service [16]. 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 [25, 1, 12]. This difficult problem is outside the scope of this paper.

5. 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 [15] to demonstrate a possible correlation between Internet routing instability and network utilization. The second set of experiments demonstrates the tool's ability to measure a large distributed system. They also show how the probe can be used to reduce application dataflows to a manageable size through on-line data reduction; and the integration of an active measurement apparatus into the Windmill infrastructure.

5.1 BGP Experiments

In these experiments, a module was created to monitor and measure the BGP [15] routing traffic exchanged between two peer border routers in order to validate one of the key observations presented in [14]. Specifically, the authors found a strong correlation between Internet routing instability and network utilization [18]. One possible reason presented 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 be modified with a UDP keepalive protocol.

BGP is an incremental routing protocol, in that upon connection establishment, two routers (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 (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. 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 [11]. The benefit of using TCP is the support for incremental routing through TCP's reliable byte-stream service. No routing data is lost between peers, obviating the need for full routing table refresh messages. 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 (infrastructure stability) is hampered at the times of high network loads. The hypothesis presented in [14] is that during peak network usage, the TCP session supporting a peer's stream of BGP keepalive messages is backed off due to congestive loss such that a message exchange between peering sessions does not complete before one of the peers' Hold Timers expires. The loss of a keepalive message prevents the delivery of subsequent keepalives until TCP can reliably deliver the lost message. This would cause peering sessions to fail at precisely those times when the network load was greatest.

Figure 3 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 do next-hop IP forwarding directly in the networking hardware. Therefore the Forwarding Cards (FC), 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

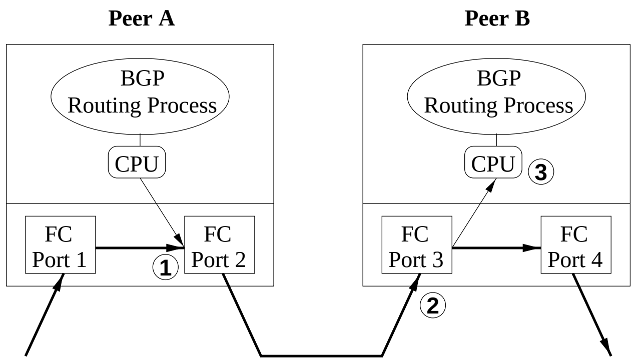

Figure 3. BGP experimental apparatus.

process runs on the router's operating system which is also managed by the CPU. In Figure 3, the overloaded route is IP data coming into Peer-A through the forwarding card handling Port-1. This data is 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 which causes the datagram to be dropped 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 [14]).

To test this hypothesis, we performed an experiment that mimics the setup described in Figure 3 on a testbed consisting of five 200 MHz Pentium-Pro Intel machines. All five were running FreeBSD 2.2.5 with 32 Megabytes 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 [21] implementation of BGP. The peering sessions' Hold Timers were set to the default 180 seconds, resulting in the generation of periodic keepalives messages at 30 second 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, was positioned "between the peers" by subscribing to the connecting Ethernet's BGP traffic (TCP source or destination ports 179) where it reconstructed the peering sessions through the use 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? Were 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? Did Windmill observe an erroneous packet?
- 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?

Figures 4 and 5 contain an abridged listing of the BGP experiment's code that reconstructs the BGP session and upon peering collapse, correlates TCP events with the termination to answer these questions. Although the figure only represents a subset of the TCP abstract protocol module's programming interface (API), it provides an illustration of Windmill's utility.

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, Windmill provides support for time-based events.

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. An example is the BGP experiment's per-direction data structure shown in Figure 4. References to this 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 are 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 bytestream service and its accompanying Data Window. We use the term *data window* to denote any unacknowledged data that has been observed by Windmill.

The abridged code shown in Figure 5 represents the incoming packet event handling routine for the BGP experiment. In addition to the `bgpHandlePacket` routine, the experiment contains two smaller procedures which are elided due to their simplicity: `bgpStart`, which subscribes to TCP flows on the well-known BGP port; and `bgpDumpOutput`, which writes statistics to a file.

```
typedef struct BGPExpState {
  int toRead;
  seqNo curPos;
  timeStamp lastKeepAliveAck;
  enum {HEADER, DATA} bgpState;
  enum {OPEN = 1, UPDATE,
        NOTIFICATION, KEEPALIVE} bgpMsgType;
} * BGPExpState_t;
```

Figure 4. BGP Analysis Module state associated with a TCP session's directional data structures.

Upon receipt, the BGP experiment passes an incoming packet to the TCP abstract protocol module by using the `tcpProcessPacket` routine shown on line 7 in Figure 5. This call causes the TCP module to execute the IP and TCP protocol as if the packet were received on the destination host. After the packet's effect on the TCP session's state has been determined, the experiment checks to see if the packet was valid on line 8. A packet is a valid TCP packet if the IP and TCP headers are valid, there are no length errors, and the checksums are correct. Lines 9-14 determine if the packet belongs to a new BGP peering session; the code that initializes the data structures has been omitted.

If the incoming packet is valid, the experiment next checks to see if it signals the termination of a BGP peering session. The experiment does this by checking whether the packet has its FIN bit set in its TCP header's flags on line 15. Currently, `tcpGetFlag` is implemented as a macro; the TCP module does not attempt to infer the current state of the end hosts' TCP state machines. Upon a TCP close, the experiment calculates an estimate of the BGP hold time (the time between the last acknowledged keepalive or update BGP message and the termination of the connection) on lines 18-23. The call to `tcpNumDataSegments` on line 26 determines how many segments of contiguous sequence numbers have been observed by Windmill, but have yet to be acknowledged by the receiver (the peer that just closed the connection). This corresponds to the data contained in the session direction's data window. The code that loops through these sequences searching for BGP keepalive messages has been elided. An experiment can get a handle on these segments through the `tcpGetDataSegInfo` command. Finally, after termination, the BGP experiment state is explicitly released in lines 32-33

When the incoming packet does not have its FIN flag set, the experiment then checks to see if the receiver's session state has acknowledged enough data to read a useful amount from the byte-stream. This is because the experiment is only reading data from the stream after it has seen an acknowledgment, therefore a packet generally triggers just one side of the session for byte-extraction. A handle to the receiver's BGP state is acquired in lines 38-39. The `tcpAvailBytes` call in line 40 returns the number of bytes in the first segment of the data window (the number of acknowledged bytes in the stream that are unread by an implicit experiment). If this is enough to parse the next portion of the current BGP message, the experiment proceeds. Depending on the portion of the message under reconstruction, the experiment will either evaluate the header or skip the data. The TCP module exports a set of procedures that are similar to the string routines defined in `<string.h>`. One of these, the `tcpncmp` command is shown on lines 43-44, is used to match the BGP header marker. These commands have been supplemented with a new class of error return values that propagate incorrect access to the data stream back to the experiment (not shown). If a match is found, the bytes are skipped (there is a similar call used to seek into the byte-stream). The TCP module provides calls that return various values from the byte-stream, examples of which are shown on lines 46-47. Additionally, the TCP module exports a peek interface to the byte stream for use when the number or type of data on the stream is uncertain. When a keepalive or update message is acknowledged by a peer, the observed time is recorded in the per-direction BGP state on lines 54-55.

Upon the collapse of a peering session, Windmill's BGP experiment dumped various statistics to disk, including a bounded window of packet contents. One such collapse is graphically represented by Figure 6. 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 circles denote TCP packets sent by Peer-A that are observed by the experiment; whereas the crosses represent those sent by Peer-B. The short vertical 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 five keepalives and their acknowledgments are measured by the experiment, however the datagram that contained the sixth keepalive is dropped. At time 390 the datagram containing the seventh keepalive is observed, which leaves a gap in the sender's TCP byte-stream that causes the receiver to reacknowledge the receipt of the fifth keepalive's sequence number. However, the TCP protocol never recovers; after 180 seconds (the value of the BGP hold-timer) the experiment observes a datagram with the FIN bit set from Peer-B, resulting in the termination of the peering session. During this experiment `tcpdump` was running on both the sender and receiver to verify Windmill's behavior. During these experiments Peer-B's `tcpdump` would occasionally observe an outbound packet that was not seen by Windmill or the `tcpdump` on Peer-A. To verify that Windmill was not dropping packets we connected seven additional machines to the Ethernet all running `tcpdump`; Windmill and all eight `tcpdumps` always agreed on their observations. Our conclusion is that these packets never made it onto the wire

```
void
bgpHandlePacket(pmfPacket *pkt, u_int handle)
{
 BGPExpState_t es;
 tcpSessDrn *sDrn;

 tcpProcessPacket(pkt);
 if (tcpValidPacket(pkt)) {
  sDrn = tcpGetSessDrn(pkt, SEND, thisExp);
  es = tcpGetExpState(sDrn);
  if (!es) {
    /* This is a new BGP session */
    bgpInitExpState(sDrn);
  }
  if (tcpGetFlag(pkt, FIN)) {
  /* Peering session severed, do Post mortum */

  /* Calculate estimated BGP hold time */
  pktGetTimeStamp(pkt, &finTime);
  sDrn = tcpGetSessDir(pkt, RECV, thisExp);
  es = tcpGetExpState(sDrn);
  holdTime = timeDiff(es->lastKeepAliveAck,
                      finTime);

  /* Determine if there are any Data gaps */
  n = tcpNumDataSegments(sDrn);

  /* ... Calls to enumerate the Data window’s
   *  segments, and check for keepalives... */

  /* Cleanup. */
  bgpFreeExpState(es);
  tcpFreeSessionDirState(sDrn);

} else {
  /* Peers still healty, check for keepalive */

  sDrn = tcpGetSessDrn(pkt, RECV, thisExp);
  es = tcpGetExpState(sDrn);
  while (tcpAvailBytes(sDrn) >= es->toRead) {
    switch (bgpState) {
      case HEADER:
        if (!tcpncmp(sDrn, tcpGetCurPos(sDrn),
            BGP_MARKER, BGP_MARKER_SIZE)) {
          tcpSkip(sDrn, BGP_MARKER_SIZE);
          es->toRead = tcpReadShort(sDrn);
          es->bgpMsgType = tcpReadByte(sDrn);
          /* ... Process Header BGP data ... */
      case DATA:
        tcpSkip(sDrn, thisExp, es->toRead);
        if ((es->bgpMsgType == KEEPALIVE ||
            (es->bgpMsgType == UPDATE)) {
          tcpGetAckTimeStamp(sDrn, seqNo,
                     es->lastKeepAliveAck);
        }
        /* ... reset scanning vars ... */
    }
```

Figure 5. Abridged version of code written for the BGP experiments.

and were dropped in Peer-B’s kernel due to the congested Ethernet. One of these packets is shown as the triangle in Figure 6 – note that Peer-A’s (the sender’s) TCP retransmits the second keepalive when it fails to receive its acknowledgment. In addition to this behavior, we have observed that in many of the experiments the TCP timeout sequence leading to the keepalive failure is more complicated than simple exponential backoff. We hope to

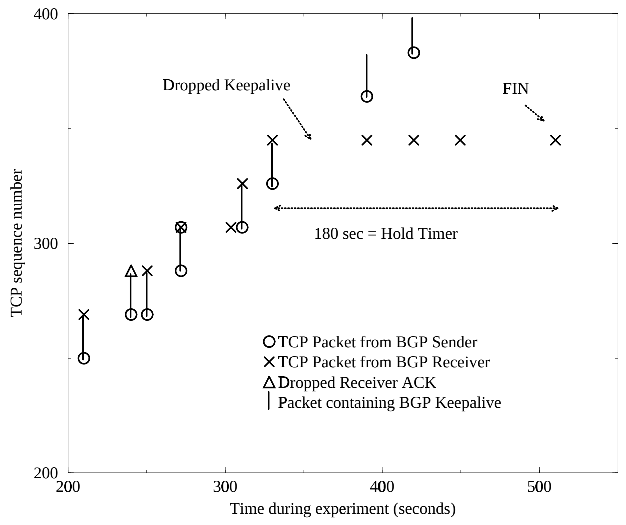

Figure 6. The collapse of a BGP peering session, due to multiple IP datagram drops in the congested TCP stream. Both peers are generating keepalive messages at 30 second intervals; the sequence numbers correspond to the TCP stream originating at the overloaded peer.

explore the dynamics in more detail in future work.

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. We are negotiating with an Internet Service Provider to place one of our probe machines at a congested peering point to further validate this occurrence. With Windmill in place, the peering session could be monitored for sustained periods.

While this result applies to BGP peering sessions at exchange points, it is perhaps more significant when applied to IBGP speakers – the Internal 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 ASes. The probability that an IBGP session’s datagrams are lost at one of these congested interior routers is increased during high network utilization, making this result 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 over normal traffic. Some routing vendors have begun to implement this. These solutions both increase the stability of BGP peering sessions and would directly strengthen the Internet infrastructure.

5.2 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, statistics from the application's data flows are extracted 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 apparatus 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 [7]. 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 weeklong experiments – occurs around the clock. The UARC system relies on a custom data distribution service [16] 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 observed all of the data communications between the main UARC server and its clients. Windmill collected the statistics by reconstructing the TCP and application-level sessions from these flows.

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 the effect a user's network connectivity was on their 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 100Mbps 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 Megabytes of 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, requests for archived data, etc. 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 uses a fixed header size with variable size data payloads. As such the reconstruction code used in the experiment is similar to the code shown in Figure 5. 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 Kbytes of statistics. The following is a subset of the questions that these measurements answered:

- Determination of the amount and level of synchronous collaboration. That is, determining 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

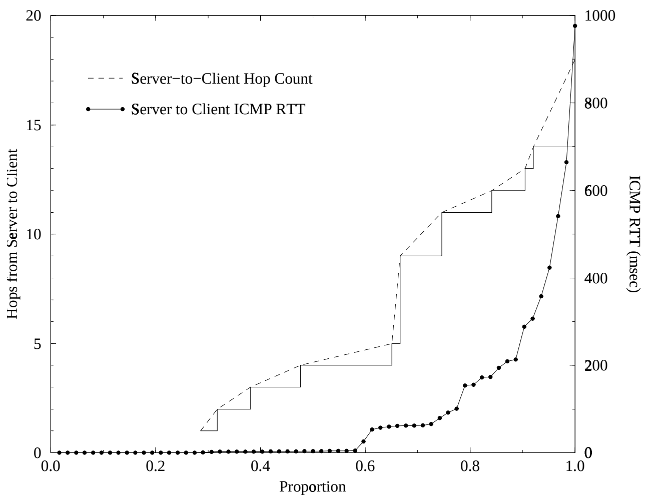

Figure 7. Cumulative distributions of initial server to client hops and ICMP round trip times.

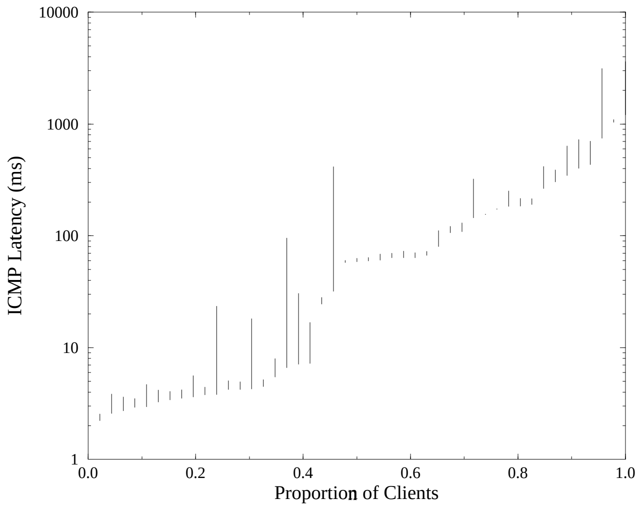

Figure 8. Server to client ICMP round trip means and variations.

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 external tool to obtain a path snapshot through a mechanism such as `traceroute`. Typically, if data is passively collected 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[22]. 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 path and ICMP round trip time measurements (in a manner similar to the `traceroute` and `ping` tools) on the client. Figure 7 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, 17% of the client hosts were five hops away from the UARC server. The distributions reflect the different types of users. Specifically, the clients one and two hops away from the server belonged to support machines located at the School of Information (where a large bank of machines for converting the real-time scientific data were located as well as the UARC system administrators); clients three hops away were located in the Psychology department (curious behavioral scientists); clients four hops away correspond to both the Michigan Space Physics and Computer Science departments' clients; clients five hops and beyond represent external space scientists.

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 for correlation 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:

Tack's represent the measured times the first acknowledgments were seen for sequence numbers *M* and *N*. The bandwidth was only estimated when there was a difference of 15 Kbytes between *N* and *M*. Since this estimate is made near the server it is similar to the rough bandwidth estimate made by FTP 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; one was a 5 hop dial-in route that had intermittent connectivity. (UARC provides support for bandwidth-constrained clients through application-level quality of service policies [16]). The experiments measured more variance in ICMP round trip times than in path length. Figure 8 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 three tiered connectivity demographic. The first tier, less than 10ms, corresponds to the clients at Michigan; the second tier, roughly between 60ms and 200ms, represents the clients well-connected to the UARC server through service providers located in the continental United States; the third tier, greater than 200ms, contains the remainder of the participants.

6. Conclusions and Future Work

This paper presented the architecture and implementation of Windmill, a passive network protocol performance tool. 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 apparatus. This work complements the efforts of several research groups that are deploying active measurement probe infrastructures within the Internet [25, 1, 12]. Unlike most tools that focus on capturing data for post analysis, Windmill was designed to support 24x7 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 Windmill Protocol Filter (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 350ns on a 200MHz Pentium-Pro. Additionally, WPF addresses limitations in past packet filtering technology by correctly handling overlapping filters. Windmill enables both 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 [14] 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 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 users' access patterns, and demonstrated the use of the passive probe to drive a complementary active measurement apparatus. These experiments highlight the ability of Windmill to measure Internet infrastructure, such as servers, without modifying end-host application or operating system code.

There are two main directions this work will be taken in the future. The first is the extension of Windmill. Specifically, it will be extended to passively measure fast serial links utilizing custom hardware splitting techniques similar to the OC3MON tool [3]. Second, Windmill will be targeted at measuring several different types of network applications including: Web servers, multiplayer games, and application-level multicast protocols.

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 for study implies the need for increased passive measurement of Internet performance. Windmill was developed for precisely this purpose[1].

7. Acknowledgments

This work was supported in part by grants from Hewlett-Packard and the National Science Foundation (NCR-9710176). Intel made the BGP experiments possible through its significant donation of lab equipment through the Technology for Education 2000 Program. Rob Malan was supported by an IBM Graduate Fellowship during part of this work. Special thanks go to Vern Paxson for his insight and patience while reading several drafts of this paper; and his extremely helpful discussions of the BGP TCP behavior. We thank Sugih Jamin and Peter Knoop for their help and cooperation. Robert Cole and Ed Perry of HP Labs provided helpful feedback during the conceptual design. We would like to thank Sally Floyd for her help with the BGP TCP analysis. Finally, we thank Gary Delp and the anonymous SIGCOMM '98 reviewers for their helpful and constructive comments.

8. References

[1] Guy Almes. Metrics and Infrastructure for IP Performance. http://io.advanced.org/csg-ippm/, September

[1] For availability see http://www.eecs.umich.edu/~rmalan

1997.

[2] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 1-14, Saint-Malo, France, October 1997.

[3] Joel Apisdorf, K. Claffy, Kevin Thompson, and Rick Wilder. OC3MON: Flexible, Affordable, High Performance Statistics Collection. In *Proceedings of INET '97*, Kuala Lumpur, Malaysia, June 1997.

[4] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkar. PathFinder: A Pattern-Based Packet Classifier. In *Proceedings of First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994.

[5] R. Caceres, P. B. Danzig, S. Jamin, and D. J. Mitzel. Characteristics of Wide-Area TCP/IP Conversations. In *Proceedings of ACM SIGCOMM '91*, September 1991.

[6] K. C. Claffy, Hans-Werner Braun, and George C. Polyzos. A parameterizable methodology for Internet traffic flow profiling. *IEEE JSAC*, 1995.

[7] C. R. Clauer, J. D. Kelly, T. J. Rosenberg, C. E. Rasmussen, E. Stauning, E. Friis-Christensen, R. J. Niciejewski, T. L. Killeen, S. B. Mende, Y. Zambre, T. E. Weymouth, A. Prakash, G. M. Olson, S. E. McDaniel, T. A. Finholt, and D. E. Atkins. A New Project to Support Scientific Collaboration Electronically. *EOS Transactions on American Geophysical Union*, 75, June 1994.

[8] Douglas E. Comer and John C. Lin. Probing TCP Implementations. In *Proceedings of the Summer USENIX Conference*, June 1994.

[9] Scott Dawson, Farnam Jahanian, and Todd Mitton. Experiments on Six Commercial TCP Implementations Using a Software Fault Injection Tool. *Journal of Software Practice and Experience*, 27(12):1385-1410, December 1997.

[10] D. Engler and M. F. Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In *Proceedings of ACM SIGCOMM '96*, August 1996.

[11] Christian Huitema. *Routing in the Internet*, pages 183–184. Prentice Hall, 1995.

[12] Internet Performance Measurement and Analysis (IPMA) project homepage. http://www.merit.edu/ipma/.

[13] Van Jacobson, Craig Leres, and Steve McCanne. tcp dump. Available via anonymous ftp from ftp.ee.lbl.gov, June 1989.

[14] Craig Labovitz, G. Robert Malan, and Farnam Jahanian. Internet Routing Instability. In *Proceedings of ACM SIGCOMM '97*, Cannes, France, September 1997.

[15] K. Lougheed and Y. Reckhter. A Border Gateway Protocol (BGP). RFC 1163, June 1990.

[16] G. Robert Malan, Farnam Jahanian, and Sushila Subramanian. Salamander: A Push-based Distribution Substrate for Internet Applications. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[17] Steve McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the 1993 Winter USENIX Technical Conference*, San Diego, January 1993.

[18] MFS Communications Mae-East Statistics Page. http://www.mfst.com/MAE/east.stats.html.

[19] Jeff C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39-51, November 1987.

[20] Jeffrey C. Mogul. Efficient Use of Workstation for Passive Monitoring of Local Area Networks. In *Proceedings of ACM SIGCOMM '90*, September 1990.

[21] Multithreaded Routing Toolkit. http://www.merit.edu/mrt/, 1996.

[22] Vern Paxson. End-to-End Routing Behavior in the Internet. In *Proceedings of ACM SIGCOMM '96*, August 1996.

[23] Vern Paxson. Automated Packet Trace Analysis of TCP Implementations. In *Proceedings of ACM SIGCOMM '97*, Cannes, France, September 1997.

[24] Vern Paxson, Guy Almes, Jamshid Mahdavi, and Matt Mathis. Framework for IP Performance Metrics. RFC 2330, May 1998.

[25] Vern Paxson, Jamshid Mahdavi, A. Adams, and Matt Mathis. An Architecture for Large-Scale Internet Measurement. *IEEE Communications*, August 1998.

[26] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. The SimOS Approach. *IEEE Parallel and Distributed Technology*, Fall 1995.

[27] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, 1995.

[28] Masanobu Yuhara, Brian Bershad, Chris Maeda, and Eliot Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of the 1994 Winter USENIX Technical Conference*, January 1994.