Monarch: A Tool to Emulate Transport Protocol Flows over the Internet at Large

Andreas Haeberlen
MPI for Software Systems, Rice University
ahae@mpi-sws.mpg.de

Marcel Dischinger
MPI for Software Systems
mdischin@mpi-sws.mpg.de

Krishna P. Gummadi
MPI for Software Systems
gummadi@mpi-sws.mpg.de

Stefan Saroiu
University of Toronto
stefan@cs.toronto.edu

This paper proposes Monarch, a novel tool that accurately emulates transport protocol flows from an end host controlled by its user to any other Internet host that responds to simple TCP, UDP, or ICMP packet probes. Since many Internet hosts and routers respond to such probes, Monarch can evaluate transport protocols, such as TCP Reno, TCP Vegas, and TCP Nice, over a large and diverse set of Internet paths. Current approaches to evaluating these protocols need control over both end hosts of an Internet path. Consequently, they are limited to a small number of paths between nodes in testbeds like PlanetLab, RON or NIMI. Monarch's ability to evaluate transport protocols with minimal support from the destination host enables many new measurement studies. We show the feasibility of using Monarch for three example studies: (a) understanding transport protocol behavior over network paths that are less explored by the research community, such as paths to cable and DSL hosts, (b) investigating the relative performance of different transport protocol designs, such as TCP Vegas and TCP Reno, and (c) testing protocol implementations under a wide range of experimental conditions.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems; C.2.2 [**Computer Systems Organization**]: Computer-Communication Networks—*Network Protocols*; C.2.5 [**Computer Systems Organization**]: Computer-Communication Networks—*Local and Wide-Area Networks*

General Terms

Experimentation, Measurement, Performance

Keywords

Emulation, transport protocols, network measurement

IMC'06, October 25–27, 2006, Rio de Janeiro, Brazil.

1. INTRODUCTION

Despite a large body of work on designing new transport protocols, such as TCP Vegas [8], TCP Nice [47], TFRC [11], or PCP [3], evaluating these protocols on the Internet at large has proved difficult. Current approaches require the protocols to be deployed at both endpoints of an Internet path. In practice, this restricts the evaluation of transport protocols to studies conducted over research testbeds, such as PlanetLab [35], RON [2], or NIMI [34]. Unfortunately, these testbeds are limited in their scale and they are not representative of the many heterogeneous network environments that constitute the Internet.

In this paper, we propose Monarch, a tool that emulates transport protocol flows from an end host controlled by its user to any other Internet host that responds to TCP, UDP, or ICMP packet probes. Since many Internet hosts and routers respond to such probes, researchers can use Monarch to evaluate transport protocols in large-scale experiments over a diverse set of Internet paths. By requiring control of just one of the two end hosts of a path, Monarch enables protocol evaluation on an unprecedented scale, over millions of Internet paths.

Monarch is based on a key observation about how transport protocols typically work: a sender transfers data to a receiver at a rate determined by the latency and loss characteristics observed by data and acknowledgment packets exchanged between the two endpoints. Monarch uses generic TCP, UDP, or ICMP probes and responses to emulate this packet exchange between a local sender and a remote receiver. We discuss which transport protocols can and cannot be emulated by Monarch in Section 2.4.

Monarch is accurate because it relies on direct online measurements. For every packet transmission in its emulated flow, Monarch sends an actual probe packet of the same size to the receiver and interprets the response packet as an incoming acknowledgment. Thus, the emulated flows are subjected to a wide range of conditions affecting real network paths, including congestion, delays, failures, or router bugs. However, as Monarch controls only one end host, it can estimate the conditions of the round-trip path but not the one-way paths. Despite this limitation, our evaluation shows that packet-level traces of flows emulated with Monarch closely match those of actual network transfers.

Monarch enhances the state of the art in transport protocol evaluation. Today researchers can use controlled envi-

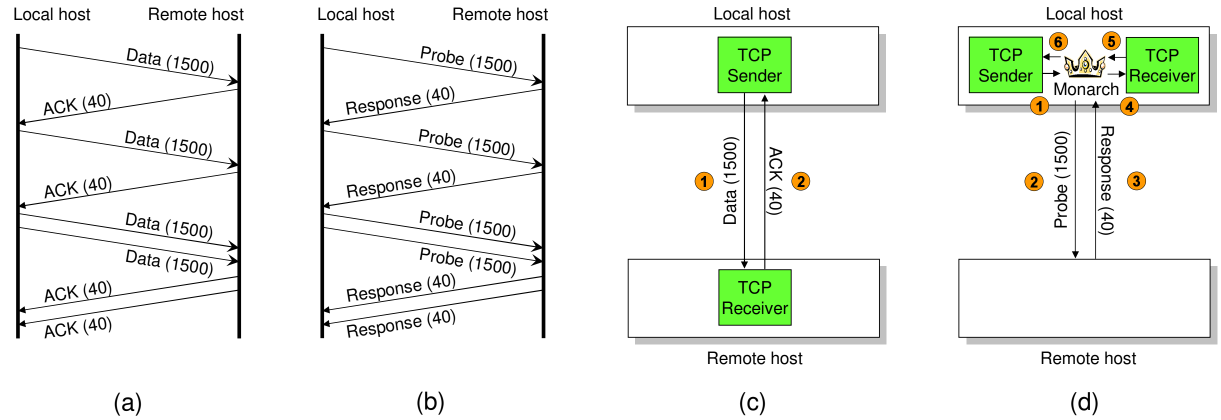


Figure 1: The Monarch packet exchange: ***In a normal TCP flow, large data packets flow in one direction and small acknowledgment packets in the other (a). Monarch emulates this by using large probe packets that elicit small responses (b). While in a normal flow, sender and receiver are on different hosts (c), Monarch colocates them on the same host and interposes between them (d). The numbers in parentheses are packet lengths in bytes.***

ronments like network emulators [46,48] or testbeds [2,34,35] for a systematic analysis of protocol behavior. Monarch complements these tools by providing live access to a real network path. This enables experiments with emerging network infrastructures, such as broadband networks, for which emulators and testbeds are not yet widely available. Further, it naturally captures the complex protocol interactions with the different configurations of networks and traffic workloads that exist in deployed systems.

In addition to capturing the behavior of transport protocols, Monarch has several benefits. Researchers can measure and gain insight into the properties of network environments less explored by the community. For example, evaluating a transport protocol over cable and DSL can provide much needed insight into the properties of broadband networks. Further, software developers can test or debug the performance and reliability of protocol implementations. These tests can uncover bugs, performance bottlenecks, or poor design decisions in the transport protocol.

The rest of the paper is organized as follows. We present the design of Monarch in Section 2, then we discuss relevant implementation details in Section 3 and evaluate Monarch's accuracy in Section 4. In Section 5, we discuss three new measurement studies enabled by Monarch. Finally, we present related work in Section 6 and summarize our conclusions in Section 7.

2. DESIGN

This section focuses on the design of Monarch. We start with an overview of how Monarch emulates transport protocols. Later, we discuss a variety of probing mechanisms Monarch can use, the number of Internet paths it can measure, the types of transport protocols it can emulate, and the factors that affect its accuracy.

2.1 How does Monarch work?

In a typical transport protocol, such as TCP, a sender on one host sends large data packets to a receiver on another host, and the receiver responds with small acknowledgment packets (Figure 1a). Monarch emulates this packet exchange by sending large *probe packets* to the remote host that elicit small responses (Figure 1b). To emulate a TCP flow, Monarch creates both a TCP sender and a TCP receiver on the *same* local host, but interposes between them (see Figure 1d). Whenever the sender transmits a packet, Monarch captures it and instead sends a probe packet of the same size to the remote host. As soon as it receives a response from the remote host, Monarch forwards the captured packet to the receiver. Packets in the reverse direction from the TCP receiver to the TCP sender are forwarded directly.

The sizes of Monarch's probe and response packets match those of TCP's data and acknowledgment packets, and they are transmitted over the same Internet paths. As a result, the sender observes similar round-trip times, queuing delays, and loss rates for its packet transmissions. Because Monarch uses online measurements as opposed to analytical models of the network, the characteristics of flows emulated by Monarch closely match those of real TCP flows.

In our simplified description above, we made several assumptions. For example, we assumed that probe packets can be matched uniquely to their response packets, that arbitrary Internet hosts would respond to probe packets, and that an accurate emulation of round-trip (rather than one-way) packet latencies and losses is sufficient for an accurate emulation of transport protocols. Later in this section, we discuss how widely these assumptions hold in the Internet at large.

Monarch's output is a packet trace similar to the output of `tcpdump`. Based on this trace, we can infer network path properties, such as packet round-trip times, and transport protocol characteristics, such as throughput. We show a particularly interesting use of this trace in Section 3.3 – Monarch can analyze its output to detect errors in its own emulated flows.

2.2 What types of probes can Monarch use?

Monarch can use several types of probe packets to emulate transport flows. It is useful to have multiple probe types to choose from because not all hosts respond to all probes. To be accurate, Monarch needs 1) the remote host to respond to *every* probe packet it receives, 2) a way to *match* responses with their corresponding probes, and 3) the sizes of the probe and response packets to be similar to those of the data and acknowledgment packets of a regular flow. Monarch currently supports the following four types of probes:

- **TCP:** Monarch's probe packet is a variable-sized TCP acknowledgment (ACK) sent to a closed port on the remote host. The remote host responds with a small, fixed size TCP reset (RST) packet. According to the TCP standard [45], the sequence number of the RST packet is set to the acknowledgment number of the probe packet header, which enables Monarch to match probes with responses.

- **UDP:** Monarch sends a variable sized UDP packet to a closed port on the remote host, which responds with a small, fixed-size ICMP 'port unreachable' message. The response packet contains the first eight bytes of the probe packet, including the IPID field of the probe packet headers [37]. By setting unique IPIDs in its probe packets, Monarch can match probes with responses.

- **ICMP echo request:** Monarch sends a variable-sized ICMP echo request ('ping') packet to the remote host, which answers with a similarly sized ICMP echo reply packet [37]. The response packet has the same sequence number field in its header as the probe packet, enabling Monarch to match probes with responses.

- **ICMP timestamp request:** Monarch sends an ICMP timestamp request message to the remote host, which answers[1] with a small, fixed size ICMP timestamp reply packet [37]. The response packet has the same sequence number field in its headers as the probe packet, enabling Monarch to match probes with responses.

These probes and responses differ in their suitability for evaluating transport protocols. For example, TCP and UDP probes allow the probe packet sizes to be varied, even as the response packet sizes are held fixed between 40 and 60 bytes. They are well suited to match the sizes of data and acknowledgment packets for many variants of the popular TCP protocol, such as Reno, Vegas, and NICE. On the other hand, the ICMP echo responses are of the same size as their probes. Consequently, they are better suited for evaluating transport flows where data flows in both directions.

2.3 How many Internet hosts respond to Monarch probes?

In theory, Monarch could emulate a transport flow to any remote host running a TCP/IP implementation, since the protocol standards require a response to each of the probes presented above. In practice, however, many hosts are either offline or behind NATs and firewalls that block or rate-limit incoming probe packets.

We conducted a simple experiment to estimate the fraction of Internet hosts that can be used as endpoints of a Monarch flow. We sent probes to three types of hosts: end hosts in commercial broadband ISPs, end hosts in academic and research networks, and Internet routers. We selected end hosts in broadband and academic networks from a 2001 trace of peers participating in the Gnutella file-sharing system [41]. We used DNS names to select hosts belonging to major DSL/cable ISPs and university domains in North America and Europe. For example, we classified a host as a BellSouth DSL host if its DNS name is of the form *adsl-*.bellsouth.net*. We discovered Internet routers by running `traceroute` to the end hosts in broadband and academic networks.

	Type of Host		
	Broadband	*Academic*	*Router*
TCP ACK	7.2 %	13.4 %	69.6 %
ICMP TsReq	18.0 %	4.9 %	63.0 %
ICMP EchoReq	25.0 %	8.9 %	89.3 %
UDP Packet	7.4 %	4.1 %	7.3 %
Any probe	28.4 %	18.1 %	90.3 %

Table 1: Fraction of Internet hosts responding to Monarch probes: ***We used three different categories with 1,000 hosts each: hosts in commercial broadband ISPs, hosts in academic and research environments, and Internet routers.***

Table 1 presents our results. We probed 1,000 hosts in each of the three host categories. Overall, more than 18% of the academic hosts, 28% of the broadband hosts, and over 90% of the routers responded to at least one of the four types of probes. While this may seem like a small percentage, there are millions of hosts in the Internet, and it should be easy to find thousands of suitable hosts for an experiment.

We believe that the primary reason for the large difference in the response rates between routers and other end hosts is the low availability of the end hosts. Unlike routers, many end hosts are often offline[2] and disconnected from the Internet. Moreover, our end hosts were selected from a trace collected five years earlier. In contrast, the router list was generated from traceroutes conducted only a few weeks before this experiment.

Using very conservative estimates, our results suggest that Monarch can evaluate transport protocols to at least 18% of Internet hosts, and to at least 7% of hosts when restricted to TCP probes only. This shows that Monarch can evaluate transport protocols over a diverse set of Internet paths, several orders of magnitude larger than what current research testbeds can provide. For example, we used Monarch to measure paths to tens of thousands of hosts in over 200 commercial cable and DSL ISPs worldwide. In contrast, research testbeds like PlanetLab have a very small number of broadband hosts.

[1] Govindan and Paxson [13] observed that some routers use a 'slow path' for generating ICMP timestamp responses, which introduces additional delays. Hence, these probes should be used with caution. We use TCP probes whenever possible.

[2] To estimate the effect of host unavailability, we probed the set of end hosts that responded to our probes for a second time after a few weeks. Only 67% of the hosts responded again, suggesting the high impact of end host unavailability.

Protocol	Usable?
TCP BIC [49], TCP Nice [47], TCP Vegas [8], TCP Westwood [22], Highspeed TCP [10], Scalable TCP [19], Fast TCP [17], PCP [3] SACK [24], FACK [23], Window scaling [15] RAP [39], TFRC [11]	Yes
ECN [38], XCP [18]	No

Table 2: Supported protocols: ***Monarch can be used to evaluate many, but not all, transport protocols.***

2.4 What transport protocols can Monarch emulate?

Monarch emulates transport protocols based on real-time, online measurements of packet latencies and losses. Hence, any transport protocol where the receiver feedback is limited to path latencies and losses can be emulated. As shown in Table 2, this includes many variants of the widely used TCP protocol, a number of protocol extensions, and several streaming protocols.

However, Monarch *cannot* emulate transport protocols that require the receiver to relay more complex information about the network to the sender. For example, Monarch cannot emulate TCP with explicit congestion notification (ECN) [38] because it would require the remote host to echo back the congestion experienced (CE) bit to the Monarch host. We are not aware of any type of probe that could be used for this purpose. Similarly, Monarch cannot be used to evaluate protocols like XCP [18] that require changes to existing network infrastructure.

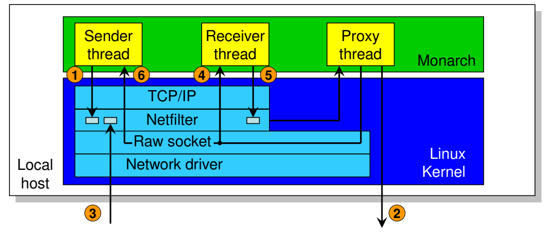
Monarch currently emulates transport flows in the downstream direction, i.e. connections in which data flows from the Monarch host to the remote host. This mimics the typical usage pattern in which an end host downloads content from an Internet server. Emulating data flows in the upstream direction from the remote host to the Monarch host requires a small probe packet that elicits a large response packet. We have not yet found a probe packet that has this property.

2.5 What factors affect Monarch's accuracy?

Monarch is based on round-trip (rather than one-way) estimates of packet latencies and losses. When packets are lost or reordered, Monarch cannot distinguish whether these events occurred on the downstream path, i.e. from the sender to the remote host, or on the upstream path, i.e. from the remote host to the sender. While this could cause Monarch flows to behave differently than regular TCP flows, our evaluation in Section 4 shows that both these events occur rarely in practice and even when they do occur, they tend to have a limited effect on Monarch's accuracy. For example, upstream packet loss and reordering affect less than 15% of all flows. Further, Monarch has a built-in self-diagnosis mechanism that can detect most of such inaccuracies using an offline analysis. Nevertheless, Monarch is not suitable for environments where upstream loss and reordering events occur frequently.

Another source of differences between Monarch and TCP flows is the delayed ACK feature. With delayed ACKs, TCP data packets received in close succession are acknowledged with a single ACK packet. In contrast, in a Monarch flow, the receiver responds to *every* probe packet, which typically doubles the number of packets flowing on the reverse path. However, because the response packets are small, this difference is likely to affect only flows experiencing severe congestion on the upstream path.

When Monarch is used on a path that contains middleboxes such as NATs or firewalls, the probes may be answered by the middleboxes rather than the end host. However, the middleboxes are often deployed close to the end host, and so the resulting loss of fidelity tends to be small. For example, Monarch probes to many commercial cable/DSL hosts are answered by the modems that are one hop away from the end host; however, the network paths to them include the 'last mile' cable or DSL links.



#	Action	Packet	Source	Destination
1	Sender transmits packet	Data	localIP	remoteIP
2	Proxy intercepts packet, saves it, and transmits a probe	Probe	localIP	remoteIP
3	Remote responds	Response	remoteIP	localIP
4	Proxy forwards saved packet to the receiver	Data	remoteIP	localIP
5	Receiver sends ACK	ACK	localIP	remoteIP
6	Proxy forwards ACK directly to the sender	ACK	remoteIP	localIP

Figure 2: Sequence of packet exchanges in Monarch implementation: ***Monarch consists of a TCP sender, a TCP receiver, and a proxy. The proxy uses Netfilter to interpose between the sender and the receiver. It applies network address translation to create the illusion that the remote host is the other endpoint of the flow.***

3. IMPLEMENTATION

In this section, we first present the details of our implementation of Monarch, which runs as a user-level application on unmodified Linux 2.4 and 2.6 kernels. We then describe how our implementation allows us to test complete, unmodified implementations of transport protocols in the Linux kernel as well as the `ns-2` simulator [30]. Finally, we discuss the self-diagnosis feature of our implementation. In particular, we show how Monarch can detect potential inaccuracies in its emulated flows by an offline analysis of its output.

3.1 Emulating a TCP flow

Our Monarch implementation uses three threads: A *sender* and a *receiver*, which perform a simple TCP transfer, as well as a *proxy*, which is responsible for intercepting packets and handling probes and responses. The proxy also records all packets sent or received by Monarch and writes them to a trace file for further analysis.

To emulate a flow to `remoteIP`, Monarch uses the Netfilter [29] framework in the Linux kernel. First, the proxy sets up a Netfilter rule that captures all packets to and from that remote address. Next, it creates a raw socket for sending packets to the remote host. Finally, the sender thread attempts to establish a TCP connection to `remoteIP`, and the packet exchange shown in Figure 2 takes place.

As usual, the sender begins by sending a SYN packet to `remoteIP` (step 1). The proxy intercepts this packet, stores it in a local buffer, and sends a similarly-sized probe packet to the remote host (step 2). The remote host responds with a packet that is also intercepted by the proxy (step 3). The proxy then looks up the corresponding packet in its buffer, modifies its destination IP address, and forwards it to the receiver (step 4). The receiver responds with a SYN/ACK packet that is captured by the proxy (step 5). The proxy then modifies its source IP address and forwards the packet back to the sender (step 6). Figure 2 also shows the details of Monarch's packet address modifications among the sender, the receiver, and the proxy.

All further packet exchanges are handled in a similar manner. If a packet transmitted to `remoteIP` is lost, its response is never received by the proxy, and the corresponding buffered packet is never forwarded to the local receiver. Similarly, reordering of packets sent to the remote host results in the buffered packets being forwarded in a different order. During long transfers, Monarch reclaims buffer space by expiring the oldest packets.

The output from Monarch includes a packet trace similar to the output of `tcpdump.` In addition, it also logs how state variables of the protocol vary over time. For example, our current implementation records TCP state variables, such as congestion window, the slowstart threshold, and the retransmission timeout, via a standard interface of the Linux kernel. The source code of our Monarch implementation is available from the Monarch web site [27].

3.2 Testing unmodified transport protocol implementations

Our proxy implementation is completely transparent to our TCP sender and TCP receiver. This is critical to Monarch's ability to test unmodified, complex protocol implementations in the Linux kernel. Further, since both the sender and the receiver run locally, we can easily evaluate the effect of different parameter choices on the sender and receiver for a given transport protocol. For example, Monarch can be used to test the sensitivity of TCP Vegas [8] to the different settings of its α and β parameters over paths to different hosts in the Internet. We can also run implementations of different TCP protocols simultaneously to understand how the protocols compete with each other. As we show in Section 5.3, this ability to test protocol implementations under a wide range of experimental conditions can be used by protocol developers to discover errors that affect the performance of their implementations.

Since it is a common practice among researchers to test new transport protocols using the `ns-2` simulator [30], we added a special interface to Monarch that allows it to connect directly to `ns-2`. Thus, researchers can conveniently use a single `ns-2` code base for both their controlled simulation and live emulation experiments. More details about this feature are available at the Monarch web site [27].

3.3 Self-diagnosis

Monarch is capable of diagnosing inaccuracies in its own emulated flows based on an analysis of its output. As we discussed earlier, the two primary factors that affect Monarch's accuracy are its inability to distinguish loss and reordering of packets on the upstream and the downstream paths, i.e., the paths from the receiver to the sender and vice-versa. These events are difficult to detect on-line, but their presence can be inferred after the emulation is finished. Monarch runs a self-diagnosis test after each emulation, which either confirms the results or lists any events that may have affected the accuracy.

3.3.1 *Detecting upstream loss and reordering*

Monarch's self-diagnosis uses the IP identifier (IPID) field in the IP headers of the response packets to distinguish between upstream and downstream events. Similar to prior techniques [6,21], Monarch's self-diagnosis relies on the fact that many Internet hosts increment the IPID field by a fixed number (typically one) for every new packet they create. However, Monarch's analysis is more involved, as it cannot send any active probes of its own and so must extract the information from a given trace.

3.3.2 *Impact of upstream loss and reordering*

Upstream packet loss and reordering events affect different transport protocols in different ways. For example, TCP Reno is more strongly influenced by packet loss than packet reordering. Even a single upstream packet loss confused as a downstream packet loss causes TCP Reno to retransmit the packet and halve its future sending rate. On the other hand, only packet reordering on a large magnitude can trigger retransmissions that affect future packet transmissions in a significant way.

Self-diagnosis tries to estimate the impact of upstream loss and reordering on Monarch flows. This impact analysis depends on the specific transport protocol being emulated. While we focus on the analysis we developed for TCP Reno, similar analysis techniques can be developed for other protocols. Our impact analysis for TCP Reno labels a flow as inaccurate if it sees an upstream packet loss or significant upstream packet reordering that causes packet retransmission. It confirms all Monarch traces that see no upstream packet loss and no significant upstream packet reordering.

We note that *confirmation* of a Monarch trace by our analysis does not imply that the trace is accurate for *all* usage scenarios. It merely suggests that the trace is likely to be accurate for *many* uses. For example, a Monarch trace that suffers only minor reordering would be confirmed. Such a trace would be accurate with respect to its throughput, latency, or packet loss characteristics, but not with respect to its reordering characteristics.

3.3.3 *Output*

After detecting upstream events and analyzing their impact, Monarch broadly classifies the result of an emulation as either confirmed, inaccurate, or indeterminate. We illustrate the decision process in Figure 3, and we discuss it below:

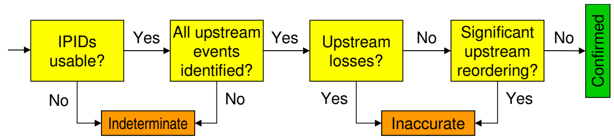


Figure 3: Self-diagnosis in Monarch: *The result is confirmed only if no known sources of inaccuracy are present.*

- **Indeterminate:** Results in this category do not contain enough information for Monarch to distinguish upstream events (loss or reordering) from downstream events in all cases. This can happen when downstream losses and upstream losses occur very close together, or when the IPIDs in the response packets are unusable because the remote host randomizes them, or sets the field to zero.
- **Inaccurate:** Monarch warns that its results could be inaccurate when it detects any upstream packet losses, *or* when the observed upstream packet reordering is significant.
- **Confirmed:** In all other cases, Monarch has not detected any upstream losses or significant reordering events. Therefore, it confirms its output.

3.3.4 Rate-limited responses

In addition to the loss and reordering analysis, Monarch also scans the entire trace for long sequences of packet losses to identify hosts that rate-limit their responses. For example, in our measurements, we observed that some hosts stop sending responses after a certain number of probes, e.g. after 200 TCP ACKs, which could be due to a firewall somewhere on the path. This pattern is easy to distinguish from packet drops due to queue overflows because in the latter case, packet losses alternate with successful transmissions. However, it is hard to distinguish losses due to path failures from end host rate-limiting.

3.4 Usage concerns and best practices

As is the case with using many active measurement tools, large-scale experiments using Monarch can raise potential security concerns. Internet hosts and ISPs could perceive Monarch's traffic as hostile and intrusive. To address this concern, Monarch includes a custom message in the payload of every probe packet. We use the message to explain the goals of our experiment, and to provide a contact e-mail address. We have conducted Monarch measurements to several thousand end hosts and routers in the Internet in hundreds of commercial ISPs over a period of seven months without raising any security alarms.

Another cause of concern is with using Monarch to send large amounts of traffic to a remote host. This can be of great inconvenience to remote hosts on broadband networks that use a per-byte payment model for traffic, where any unsolicited traffic costs the host's owner real money. To mitigate this concern, we only measure hosts in broadband ISPs that offer flat rate payment plans. In addition, we never transfer more than a few dozen megabytes of data to any single Internet host.

Finally, we would like to point out that Monarch flows compete fairly with ongoing Internet traffic as long as the emulated transport protocols are TCP-friendly.

	PlanetLab	*Broadband*	*Router*
Sender nodes	4	4	4
Receiver nodes	356	4,805	697
Successful measurements	12,166	15,642	2,776

Table 3: Traces used for our Monarch evaluation: *For each trace, we used geographically dispersed sender nodes in Seattle (WA), Houston (TX), Cambridge (MA), and Saarbrücken (Germany).*

4. EVALUATION

In this section, we present three experiments that evaluate Monarch's ability to emulate transport protocol flows. First, we evaluate the accuracy of its emulated flows, i.e., we verify how closely the characteristics of Monarch flows match those of actual TCP flows. Second, we identify the major factors and network conditions that contribute to inaccuracies in Monarch's emulations, and show that Monarch's self-diagnosis can accurately quantify these factors. Third, we characterize the prevalence of these factors over the Internet at large.

4.1 Methodology

Evaluating Monarch's accuracy over the Internet at scale is difficult. To evaluate Monarch, we need to compare its emulated flows to real transport flows over the same Internet paths. Unfortunately, generating real transport flows requires control over both end hosts of an Internet path. In practice, this would limit our evaluation to Internet testbeds, such as PlanetLab. We deal with this limitation using the following three-step evaluation:

1. In Section 4.2, we evaluate Monarch over the PlanetLab testbed. We generate both Monarch flows and real TCP flows, identify potential sources of error, and study how they affect accuracy.
2. In Section 4.3, we show that Monarch's offline self-diagnosis can accurately detect these errors from its own traces.
3. In Section 4.4, we use this self-diagnosis capability to estimate the likelihood of error in Monarch measurements over a wide variety of Internet paths.

4.1.1 Data collection

We used Monarch to emulate transport flows over three types of Internet paths: (a) paths to PlanetLab nodes, (b) paths to Internet hosts located in commercial broadband ISPs, and (c) paths to Internet routers. Table 3 shows statistics about the three datasets we gathered. All measurements involved 500kB data transfers. The TCP senders were located in four geographically distributed locations, three (Seattle, Houston and Cambridge) in the U.S. and one (Saarbrücken) in Germany, while the receivers included PlanetLab nodes, broadband hosts, and Internet routers. While gathering the PlanetLab dataset, we controlled both endpoints of the Internet paths measured, so we generated both Monarch and normal TCP flows. In the other two datasets we only controlled one endpoint, so we generated only Monarch flows.

	DSL						***Cable***				
	Ameritech	*BellSouth*	*BTOpen World*	*PacBell*	*Qwest*	*SWBell*	*Charter*	*Chello*	*Comcast*	*Roadrunner*	*Rogers*
Company	AT&T	BellSouth	BT Group	AT&T	Qwest	AT&T	Charter Comm.	UPC	Comcast	TimeWarner	Rogers
Region	S+SW USA	SE USA	UK	S+SW USA	W USA	S+SW USA	USA	Holland	USA	USA	Canada
Hosts Measured	214	242	218	342	131	1,176	210	295	856	997	124
Offered BWs (bps)	384K, 1.5M, 3M, 6M	256K, 1.5M, 3M, 6M	2M	384K, 1.5M, 3M, 6M	256K, 1.5M, 3M, 6M	384K, 1.5M, 3M, 6M	3M	384K, 1M, 3M, 8M	4-8M	5M, 8M	128K, 1M, 3M, 6M

Table 4: Broadband trace statistics: The statistics are broken down by ISP; *offered BWs* refers to the access link capacity advertised on the ISPs' web sites.

Our *PlanetLab* measurements used 356 PlanetLab nodes world-wide as receivers. To each PlanetLab node, we conducted five data transfers using Monarch interspersed with five normal TCP transfers in close succession[3], for a total of ten data transfers from the each of the senders in Seattle, Houston, Cambridge, and Saarbrücken. We ran `tcpdump` on both the sending and the receiving node to record packet transmissions in either direction.

Our *Broadband* measurements used 4,805 cable and DSL end hosts in 11 major commercial broadband providers in North America and Europe. The list of broadband ISPs we measured is shown in Table 4. We selected these hosts by probing hosts measured in a previous study of Napster and Gnutella [41]. For each host that responded, we used its DNS name to identify its ISP.

Our *Router* measurements used $1,000$ Internet routers we discovered by running `traceroute` to hosts in the broadband data set. Only 697 of these routers responded to Monarch's probe packets.

These three datasets are available at the Monarch web site [27].

4.2 Accuracy over PlanetLab

In this section, we compare the behavior of Monarch flows to TCP flows on Internet paths to PlanetLab nodes. We focus on two different aspects. First, we investigate whether the packet-level characteristics of the emulated flows closely match those of TCP flows. For this, we analyze the sizes and transmission times of individual packets. Second, we compare their higher-level flow characteristics, such as throughput and overall loss rate.

4.2.1 Packet-level characteristics

Monarch emulates transport protocol flows at the granularity of individual packet transmissions. In this section, we compare the packet-level characteristics of Monarch and TCP flows to show that they closely match in terms of number of packets, sizes of packets, packet transmission times, and evolution of important protocol state variables.

We begin by comparing two flows on a single Internet path: one emulated with Monarch and one actual TCP flow. Figure 4(a) shows the times when individual data segments were transmitted. The graph shows that the transmission times of packets in the Monarch flow are almost indistinguishable from those in the TCP flow.

Next, we examine how TCP protocol state variables change during the flows. Figure 4(b) shows a plot of the congestion window (CWND) and the retransmission timeout (RTO) for both flows. This information is recorded in Monarch's output trace using the `TCP_INFO` socket option. The CWND plot shows the typical phases of a TCP flow, such as slowstart and congestion avoidance. Both flows went through these phases in exactly the same way. The TCP variables of the Monarch flow closely match those of the actual TCP flow, suggesting a highly accurate Monarch emulation.

Next, we compare the properties of aggregate data from all our PlanetLab traces. We begin by comparing the number of packets sent and received in the Monarch flows and their corresponding TCP flows. Figure 5 shows the relative difference for each direction, using the number of packets in the TCP flow as a basis. 65% of all flow pairs sent the same number of packets; the difference is less than 5% of the packets for 93% of all flow pairs. This is expected because (a) both Monarch and TCP use packets of the same size, and (b) both flows transfer the same 500kB of data. Moreover, when we compare only flows with no packet losses, the difference disappears entirely (not shown). This suggests that packet losses account for most of the inaccuracies in the number of packets sent.

Figure 5 shows a substantial difference in the number of packets received in the upstream direction. This is due to delayed ACKs: TCP flows acknowledge several downstream packets with a single upstream packet, while Monarch flows contain one response packet for every probe. However, acknowledgment packets are small (typically 40 bytes), and we will show later that this additional traffic has little impact on high-level flow characteristics, such as throughput.

Finally, we repeat our analysis of packet transmission times on a larger scale, across all PlanetLab traces. Our goal is to check whether the rates and times at which packets are transmitted are similar for TCP and Monarch flows.

We compare the times taken to transfer 10%, 30%, 50%, 70%, and 90% of all bytes in the 500kB transfer. Figure 6 shows the difference between times taken to complete Monarch and TCP traces relative to the TCP traces. The error stays small for every part of the transfer, suggesting that packets are sent out at similar rates during the flows' lifetimes.

[3] We also ran the Monarch and TCP flows concurrently, and compared them. The results were similar to those we obtained when we ran Monarch and TCP alternately in close succession. Hence, we show only results from the latter.

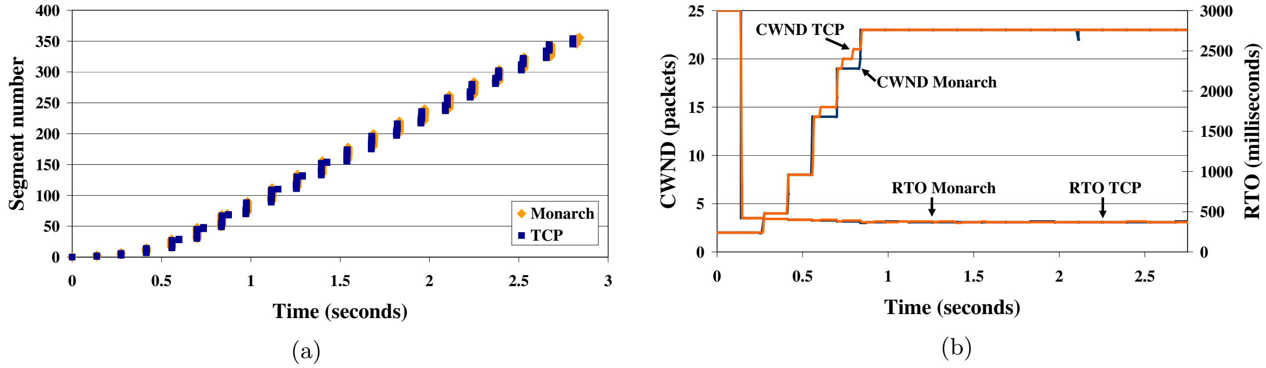


Figure 4: Comparison between typical Monarch and TCP flows: *The figures show when each segments was sent (a) and how the congestion window and the retransmission timeout evolved over time (b). The plots for Monarch and TCP are so close that they are almost indistinguishable.*

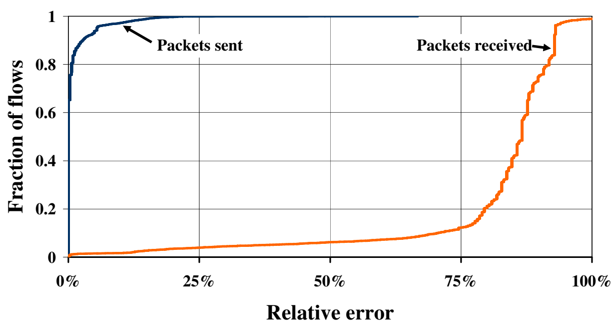


Figure 5: Traffic generated by Monarch and TCP: *Relative difference between the number of packets sent and received, shown as cumulative distributions. Positive values indicate that Monarch sent/received more packets than TCP.*

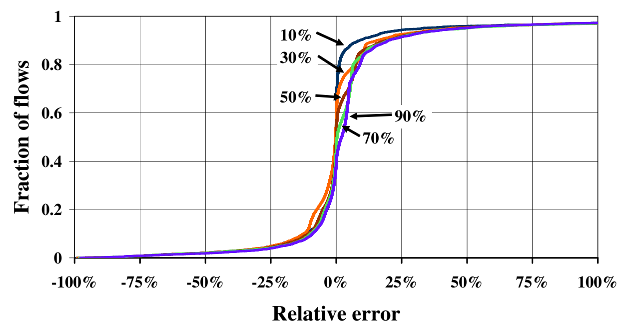


Figure 6: Progress of TCP and Monarch flows: *Relative error of the time it took to complete a certain fraction of the transfer between pairs of TCP and Monarch flows, shown as a cumulative distribution.*

To summarize, we find that Monarch and TCP flows match with respect to several packet-level characteristics, including the number and sizes of packets sent, the evolution of important protocol state variables, and the transmission times of individual segments.

4.2.2 Flow-level characteristics

In this section, we investigate whether Monarch and TCP traces are similar with respect to several high-level flow characteristics, such as throughput, round-trip times, queueing delay, and packet loss.

Throughput: Figure 7 shows the cumulative distributions of the throughput for Monarch and TCP flows. While the lines for Monarch and TCP match well, Monarch flows tend to have a slightly lower throughput than TCP flows. The figure also shows a second pair of lines, which uses only flows without packet losses and retransmissions. Interestingly, these lines show almost no difference between Monarch and TCP. This suggests that the small errors in Monarch's throughput estimates might be due to packet losses.

To quantify this error, Figure 8 shows the relative throughput difference in pairs of consecutive Monarch and TCP flows, using TCP's throughput as a base (recall that we took ten measurements on each path, alternating between Monarch and real TCP flows). In over 50% of the flow pairs, the throughput of the Monarch flow differs from the throughput of the TCP flow by less than 5%, which is a good match. However, not all these differences are due to inaccuracies in Monarch. Figure 8 also shows the throughput differences between two consecutive TCP flows along the same paths. The two plots are similar, suggesting that the dominant cause of these differences is unstationarity in the network, e.g., fluctuations in the amount of competing traffic.

Thus, while packet losses can cause Monarch to underestimate the throughput in general, their impact is fairly small, often smaller than the impact of the unstationarity in network path properties during the course of the flow.

Latency: Next, we focus on the latencies and delays experienced by packets during Monarch and TCP flows. We compute three types of packet latencies or round-trip times (RTT): minimum RTT, maximum RTT, and queueing delay. To remove outliers, we take the maximum RTT to be the 95^{th} percentile of all packet RTTs, and compute the queueing delay as the difference between maximum and minimum RTTs. Figures 9 shows the difference in the estimates of RTTs between Monarch and TCP traces, as a percentage of TCP estimates. We also show how estimates from successive measurements of TCP flows differ from each other.

There are two take-away points from Figure 9. First, Monarch's estimates of minimum and maximum RTT closely match the TCP estimates. In fact, Monarch's errors are indistinguishable from the variation observed in the

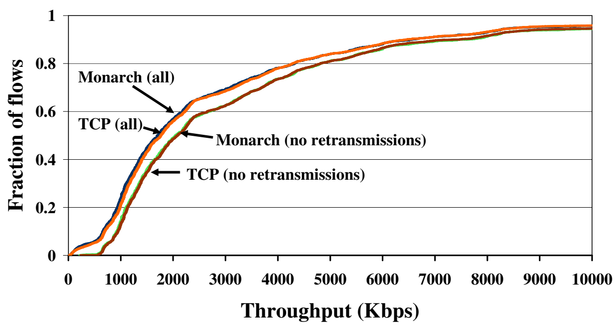


Figure 7: Throughput comparison between Monarch and TCP flows: *The cumulative distributions are very close; if only flows without packet retransmissions are considered, the distributions are indistinguishable.*

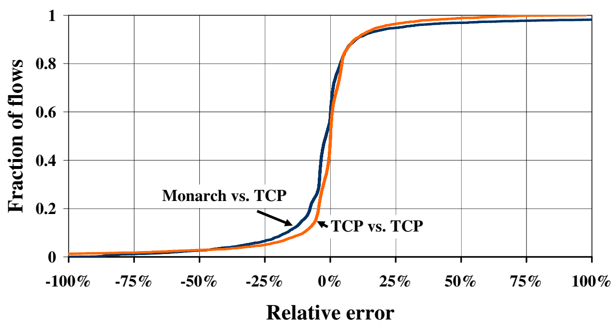


Figure 8: Relative throughput error between pairs of TCP and Monarch flows, and between pairs of TCP flows: *The error is similar, which suggests that its primary cause is unstationarity in the network, and not a problem with the Monarch emulation.*

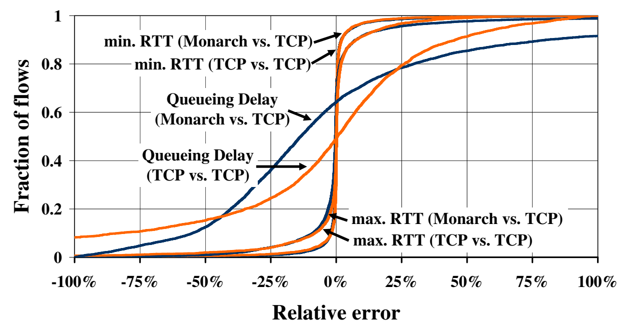


Figure 9: Relative RTT difference between successive TCP and Monarch flows: *The error in extremely small, except for the queueing delay. Queueing delay was generally low in the PlanetLab trace, which is why even small variations lead to big relative errors.*

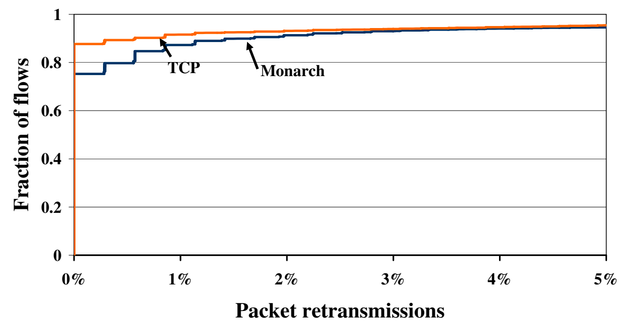


Figure 10: Retransmissions per flow for Monarch and TCP: *Monarch shows more retransmissions because it must retransmit packets for losses in both upstream and downstream directions, while TCP needs to retransmit only packets lost on the downstream.*

estimates between successive TCP measurements along the same paths. This points to the efficiency of our Monarch implementation; despite the additional packet processing overhead in our interposing proxy, we add negligible overhead to the packet latencies. Second, queueing delays show a much larger variation or unstationarity over time compared to minimum and maximum RTTs. The reason for these large relative differences is that the absolute values are very low. Over 76% of queueing delay estimates are below 10 milliseconds. Hence, even a small 1-millisecond variation corresponds to a 10% difference.

Packet loss: Finally, we investigate the loss rates in the flows. We note that both Monarch and TCP senders retransmit packets that they *perceive* to be lost, which might be different from the packets that were *actually* lost. For example, TCP might mistake massive packet reordering for a loss and trigger a retransmission. Our interest here is in the perceived loss rates of these flows, so we use the packet retransmission rate for loss rate.

Figure 10 shows cumulative distributions of retransmission rates for both Monarch and TCP flows. 75% of all Monarch flows and 88% of all TCP flows do not contain any retransmissions and therefore do not perceive packet loss. Thus, packet retransmissions do not affect a majority of both Monarch and TCP flows. Of the flows that do contain retransmissions, Monarch shows a clearly higher retransmission rate than TCP. This is expected because Monarch flows must retransmit packets for losses in both upstream and downstream directions, while TCP needs to retransmit only packets lost on the downstream, due to cumulative acknowledgments.

Summary: Our analysis shows that Monarch can accurately emulate TCP flows with respect to flow-level properties such as throughput, latency, and queueing delay. However, Monarch's inability to distinguish between upstream and downstream packet loss causes it to over-estimate packet loss. The impact of this inaccuracy is limited to the small fraction of flows that see upstream packet loss.

4.3 Reliability of self-diagnosis

In the previous section, we showed that the primary source of inaccuracy in a Monarch emulation is upstream packet loss. In this section, our goal is to show that Monarch's self-diagnosis feature (Section 3.3) can reliably detect upstream packet loss, and thus, warn the user of potential inaccuracies.

We tested this feature on the Monarch flows in our PlanetLab trace. For each flow, we compared the `tcpdump` traces from the sender and the receiver to determine how many packets had actually been lost on the downstream and the upstream. Then we compared the results to the output of Monarch's self-diagnosis for that flow; recall that this uses only the sender-side trace.

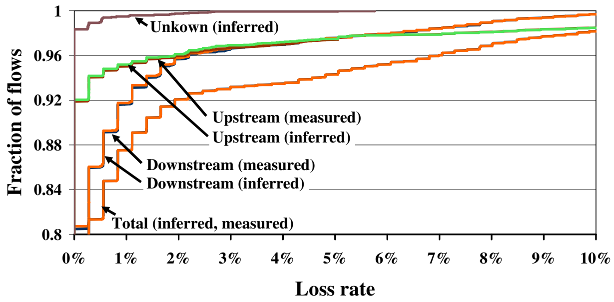


Figure 11: Self-diagnosis is accurate: *The number of downstream and upstream losses inferred by Monarch's self-diagnosis matches the actual measurements. Only a small number of losses cannot be classified as either downstream or upstream.*

Result	*Broadband*		*Router*	
Confirmed	13,168	84.2 %	2,317	83.5 %
Inaccuracte	1,130	7.2 %	164	5.9 %
Indeterminate	1,344	8.6 %	295	10.6 %
Traces total	15,642	100.0 %	2,776	100.0 %

Table 5: Monarch is accurate over real Internet paths: *In the majority of the flows, self-diagnosis did not detect any inaccuracies. Note that even when an inaccuracy is reported, its impact on flow-level metrics such as throughput may be quite small.*

Figure 11 shows the results. Self-diagnosis could not distinguish between all upstream and downstream losses (see Section 3.3) for a very small number of flows (less than 2%). In these cases, Monarch printed a warning. For the majority of flows for which self-diagnosis could infer the loss rates, the measured and the inferred loss rates match extremely well in both upstream and downstream directions. As expected, the total loss rate plots are identical.

We conclude that Monarch's self-diagnosis can reliably detect the major source of inaccuracy in an emulated flow.

4.4 Accuracy over the Internet at large

In the previous two sections, we showed that upstream loss is the most important source of inaccuracies in Monarch emulations, and that Monarch's self-diagnosis can reliably detect the presence of upstream loss. Our goal in this section is to show that upstream losses are rare even when Monarch is used over real Internet paths.

We ran Monarch's self-diagnosis over our two Internet traces; the first trace consists of 15,642 flows to 4,805 broadband hosts, and the second trace contains 2,776 flows to 697 Internet routers. Table 5 summarizes our results. About 10% of the traces could not be analyzed by Monarch. In the broadband dataset, 7.1% of the traces did not contain usable IPIDs (8.3% in the router dataset), and 1.5% (2.3%) contained a loss that could not be classified as either upstream or downstream. In either of these cases, self-diagnosis was aborted immediately.

Overall, 84.2% of the broadband traces and 83.5% of the router traces were confirmed by self-diagnosis because neither upstream losses nor significant reordering errors were detected. This includes the 15.8% (24.9%) of the traces that contained only minor reordering errors that would not have changed the number of duplicate ACKs, and therefore would not have affected any packet transmissions. Only 7.2% of the broadband traces were reported as inaccurate; for the router traces, the fraction was only 5.9%.

We conclude that a majority of our flows to Internet hosts did not suffer from upstream packet loss or significant reordering, the two primary sources of inaccuracy in Monarch. This suggests that Monarch can be used to accurately emulate TCP flows to a large number of Internet hosts. Moreover, our results show that the IPID-based self-diagnosis is applicable in most cases.

4.5 Summary

In this section, we showed that Monarch is accurate: its emulated TCP flows behave similarly to real TCP flows with respect to both packet-level and flow-level metrics. We also showed that the most important source of error in Monarch's flows is upstream packet loss, and that this can be reliably detected by Monarch's built-in self-diagnosis. Further, our examination of large sets of Monarch flows to various Internet hosts, including hundreds of routers and thousands of broadband hosts, revealed that less than 10% of these flows suffer from upstream packet loss. From this, we conclude that Monarch can accurately emulate TCP flows to a large number of Internet hosts.

5. APPLICATIONS

Monarch's ability to evaluate transport protocol designs over large portions of the Internet enables new measurement studies and applications. We used Monarch to conduct three different types of measurement experiments. In this section, we describe these experiments and present some preliminary results from them to illustrate their potential benefits.

5.1 Evaluating different transport protocols

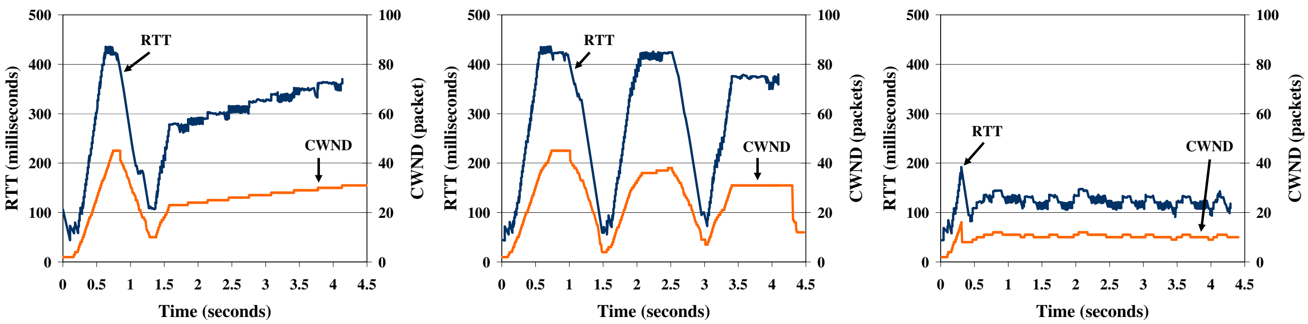
New transport protocol designs [3, 10, 47, 49] continue to be proposed as the Internet and its workloads change over time. However, even extensive simulation-based evaluations face skepticism whether their results would translate to the real world. The resulting uncertainty around how well these protocols would compete with existing deployed protocols hinders their actual deployment. With Monarch, researchers can evaluate their new protocol designs over actual Internet paths.

We used Monarch to compare three different TCP congestion control algorithms implemented [4] in the Linux 2.6.16.11 kernel: NewReno [12], BIC [49], and Vegas [8]. In our experiment, we emulated 500kB data transfers from a local machine to several hosts in broadband (cable and DSL) ISPs, using each of the three congestion control algorithms in turn.[5] We examined the traces generated by Monarch for differences in protocol behavior.

Figure 12 shows the difference between the algorithms over a single, but typical path. The graphs show how the congestion window (CWND) and the round-trip time (RTT)

[4]Note that the Linux implementation of TCP protocols may differ significantly from their reference implementation or their standard specification.

[5]In Linux 2.6 kernels, it is possible to switch between different TCP congestion control algorithms at runtime.



(a) (b) (c)

Figure 12: Comparing the performance of different TCP protocols over an Internet path between a host in Germany and a host in the BTOpenWorld DSL network: *variation of packet round-trip times (RTT) and the congestion window (CWND) over the duration of a flow using (a) NewReno, (b) BIC, and (c) Vegas. The steep drops in RTT and CWND values are due to packet losses. Compared to Reno, BIC shows higher RTTs and losses, while Vegas shows lower RTTs and losses.*

evolve over the duration of the transfer. All flows begin in the slow-start phase, where the CWND increases rapidly until the flow loses a packet and enters the congestion avoidance phase. The TCP NewReno graph shows that the RTT increases from 44ms at the beginning to well over 300ms before it loses a packet. This suggests the presence of a long router queue at the congested link on this broadband Internet path. TCP BIC, which has been adopted as the default TCP protocol by Linux since kernel version 2.6.7, shows a similar pattern but ramps up the congestion window much faster after each loss, which results in even higher queueing delays and packet losses. In contrast to NewReno and BIC, Vegas enters a stable state with a round-trip time of about 100ms without suffering a single loss.

Our experiment shows that TCP BIC, the default congestion control algorithm in Linux, exhibits the worst performance both in terms of packet delay and packet loss. This is not particularly surprising because BIC is designed for Internet paths that have a high bandwidth-delay product. In contrast, our measurement path includes a broadband link with relatively low bandwidth. However, since many hosts today use broadband Internet connections, it might be important to improve BIC's performance over broadband networks.

Our Monarch results, while preliminary, show the importance of understanding the behavior of new protocols over a variety of real network paths before deploying them widely.

5.2 Inferring network path properties

Monarch can be used to infer properties of network paths that have received relatively little attention by the research community, such as paths to end hosts in commercial cable and DSL networks. For this study, we analyzed the Broadband trace described in Section 4.1. This trace contains Monarch flows to 4,805 broadband hosts in 11 major cable and DSL networks in North America and Europe. Our analysis inferred several path properties, such as throughput, round-trip times, queueing delays, loss, and reordering. While a detailed characterization of these properties is beyond the scope of this paper, we present some initial results about the overall throughput of these flows and discuss their potential implications.

Figure 13 shows the distribution of throughput for flows to different cable and DSL ISPs. The throughput plots for the DSL ISPs jump sharply at 256Kbps, 384Kbps, 512Kbps and 1Mbps. These data rates roughly correspond to the link speeds advertised by these ISPs (see Table 4), which indicates that our transfers were able to saturate the access link, which is likely to be the bottleneck. However, the throughput for cable flows do not exhibit similar jumps, even though cable ISPs advertise discrete link speeds as well. This suggests that DSL flows and cable flows may be limited by different factors; access link capacities seem to affect DSL flows to a greater extent than cable flows.

Overall, our experiment demonstrates how Monarch can be used to infer properties of network paths that have proven difficult to measure in the past. Previous studies of broadband hosts [20] required control over both endpoints, and consequently were limited to a small number of broadband paths.

5.3 Testing complex protocol implementations

Modern transport protocols (e.g. TCP NewReno with fast retransmit and recovery) are so complex that it is often difficult to implement them correctly. While program analysis techniques [28] could help debug functionally incorrect implementations, it is important to test the performance of these protocols in the real world to find performance problems. Monarch is particularly useful for testing protocols because it can run complete and unmodified protocol implementations.

We used Monarch to emulate TCP flows to several different types of hosts, including broadband hosts and academic hosts. In this process, we discovered bugs in the Linux TCP stack that tend to manifest themselves frequently over certain types of Internet paths. For example, we found that the Linux 2.6.11 implementation of Fast Recovery [12] can cause the congestion window to collapse almost entirely, instead of merely halving it. This problem can severely reduce throughput, and it occurs repeatedly over paths to DSL or cable hosts.

The purpose of Fast Recovery is to allow the TCP sender to continue transmitting while it waits for a retransmitted segment to be acknowledged. Linux uses a variant known as rate halving [43], which transmits one new segment for

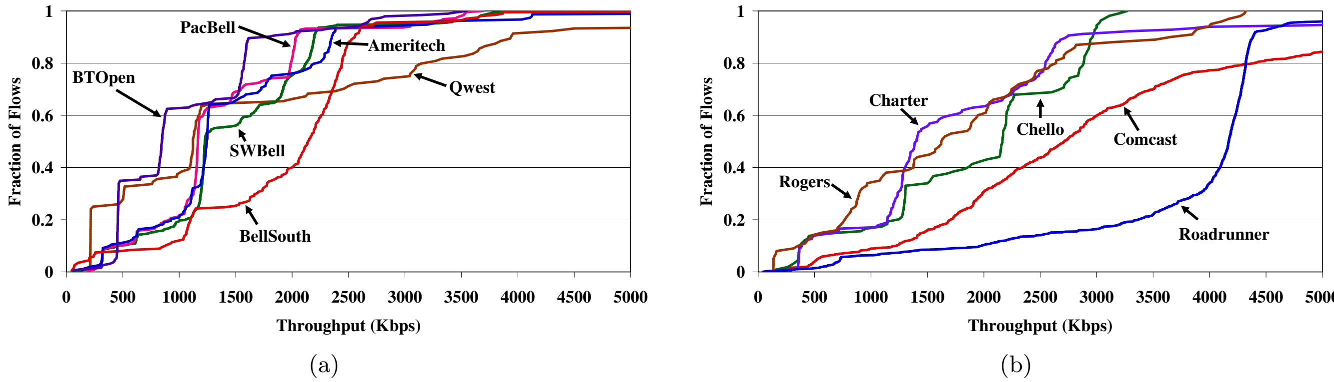


Figure 13: Broadband flows: ***Cumulative distributions of TCP flow throughput achieved over major (a) DSL and (b) cable access networks. Notice how DSL throughput raises sharply at discrete bandwidth levels while cable throughput does not.***

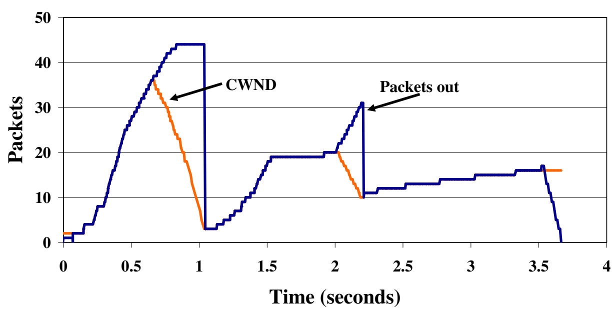


Figure 14: Incorrect rate halving in Linux TCP: ***After the first loss, the congestion window falls below half its original value.***

every other ACK received. Thus, one new packet is sent for every two packets that leave the network. Under normal conditions, this has the effect of gradually decreasing the number of packets in flight by half. Linux 2.6.11 implements rate halving by estimating the number of packets in flight, and capping the congestion window at that number.

However, we found that this approach fails when the congestion window approaches the send buffer size. Figure 14 shows an example of a flow that saw its first loss after 0.6 seconds, when the congestion window was 36 packets wide. Initially, Linux was able to send 8 additional segments for every other ACK as expected. But, once it reached the default send buffer size of 64kB (44 packets), it could not transmit more new segments. After this point, with no new segments being transmitted, the number of packets in flight, and consequently the congestion window, decreased rapidly. *Every* incoming ACK reduced the congestion window by one packet, causing it to fall far below the slowstart threshold of 18 packets. Thus, after leaving Fast Recovery, Linux fell back into slowstart for over half a second. Note that a second loss at 2.0 seconds was handled correctly because the congestion window was still fairly small.

Monarch helped us discover this problem because it allowed us to test the complete and unmodified protocol implementation (in this case, the NewReno code in the Linux kernel) over a wide range of real links with different characteristics.

6. RELATED WORK

Monarch leverages existing protocols in unanticipated ways to perform measurements that were previously intractable. This approach is similar to several other measurement tools. Sting [44] manipulates the TCP protocol to measure packet loss. T-BIT [25, 32] exploits the TCP protocol to characterize Web servers' TCP behavior. King [14] uses DNS queries to measure latencies between two arbitrary DNS servers. SProbe [42] sends packet pairs of TCP SYN packets to measure bottleneck bandwidth to uncooperative Internet hosts. Like Monarch, these tools send carefully crafted packet probes to remote Internet hosts to measure network properties.

There is a large body of literature on evaluating transport protocol designs and implementations. Much of the previous work relies one of following three approaches to characterize protocol behavior. The first approach uses synthetic network simulators and emulators, such as ns-2 [30], NetPath [1], dummynet [40], NIST [9], and ModelNet [46]. There is an impressive amount of research on protocol modeling and characterization in these controlled environments. Padhye [31] discusses a summary of papers on TCP modeling. Unlike Monarch, previous simulators and emulators either use analytical models to generate TCP traffic or they simulate different synthetic network environments.

The second approach of evaluating transport protocols is based on active measurement. Bolot [7] and Paxson [33] performed some of the initial studies on network packet dynamics along a fix set of Internet paths. Padhye and Floyd [32] characterized the TCP behavior of a large set of popular Web servers. Medina et al. [25] investigated the behavior of TCP implementations and extensions. In a different project, Medina et al. [26] characterized the effect of network middleboxes on transport protocols. More recently, several studies have used PlanetLab [36] to examine different aspects of TCP network traffic.

The third approach of evaluating transport protocols relies on passive measurements. Based on the traces of TCP flows to a busy Web server, Balakrishnan et al. [5] presented a detailed analysis of the performance of individual TCP flows carrying Web traffic. Jaiswal et al. [16] used traffic traces of a Tier-1 ISP to investigate the evolution of a TCP connection variables over the lifetime of a TCP connection. More recently, Arlitt et al. [4] have used Web traces to investigate the impact of latency on short transfers' durations.

7. CONCLUSIONS

In this paper, we presented Monarch, a tool that emulates transport protocol flows over live Internet paths. Monarch enables transport protocols to be evaluated in realistic environments, which complement the controlled environments provided by the state of the art network simulators, emulators or testbeds. Monarch is highly accurate: its emulated flows closely resemble TCP flows in terms of throughput, loss rate, queueing delay, and several other characteristics.

Monarch uses generic TCP, UDP, or ICMP probes to emulate transport protocol flows to any remote host that responds to such probes. By relying on minimal support from the remote host, Monarch enables protocols to be evaluated on an unprecedented scale, over millions of Internet paths.

We used Monarch for three novel experiments. First, our preliminary study on the performance of different congestion control algorithms (TCP Reno, TCP Vegas and TCP BIC) shows that much remains to be understood about the behavior of even widely adopted protocols over the Internet at large. Second, we showed that Monarch measurements can be used to infer network properties of less-studied Internet paths, such as paths to cable and DSL hosts. Third, we used Monarch to test complete and unmodified TCP protocol implementations in the Linux kernel over a variety of Internet paths, and we discovered nontrivial bugs. Based on our experience, we believe that Monarch can help the research community conduct large-scale experiments leading to new insights and findings in the design and evolution of Internet transport protocols.

8. ACKNOWLEDGMENTS

We would like to thank Steve Gribble, Dan Sandler, and Emil Sit for generously hosting Monarch servers for our experiments. Peter Druschel and our anonymous reviewers provided detailed and helpful feedback on the earlier versions of this draft. The `ns-2` interface for Monarch was developed by Prateek Singhal.

9. REFERENCES

[1] S. Agarwal, J. Sommers, and P. Barford. Scalable network path emulation. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Washington, DC, 2005.

[2] D. G. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Experience with an Evolving Overlay Network Testbed. *ACM Computer Communication Review*, 33(3), July 2003.

[3] T. Anderson, A. Collins, A. Krishnamurthy, and J. Zahorjan. PCP: Efficient endpoint congestion control. In *Proceedings of NSDI'06*, May 2006.

[4] M. Arlitt, B. Krishnamurthy, and J. C. Mogul. Predicting short-transfer latency from TCP Arcana: A trace-based validation. In *Proceedings of Internet Measurement Conference*, Berkeley, CA, October 2005.

[5] H. Balakrishnan, V. N. Padmanbhan, S. Seshan, M. Stemm, and R. H. Katz. TCP behavior of a busy Internet server: Analysis and improvements. In *Proceedings of IEEE Infocom*, San Francisco, CA, USA, March 1998.

[6] J. Bellardo and S. Savage. Measuring packet reordering. In *Proceedings of the 2002 ACM SIGCOMM Internet Measurement Workshop (IMW)*, Marseille, France, November 2002.

[7] J.-C. Bolot. Characterizing end-to-end packet delay and loss in the Internet. In *Proceedings of ACM SIGCOMM*, San Francisco, CA, September 1993.

[8] L. S. Brakmo and L. Peterson. TCP Vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communication*, 13(8):1465–1480, October 1995.

[9] M. Carson and D. Santay. NIST Net – a Linux-based network emulation tool. *SIGCOMM Computer Communications Review*, 33(3):111–126, 2003.

[10] S. Floyd. RFC 3649 - HighSpeed TCP for large congestion windows, Dec 2003. `ftp://ftp.rfc-editor.org/in-notes/rfc3649.txt`.

[11] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. In *Proceedings of SIGCOMM'00*, Aug 2000.

[12] S. Floyd, T. Henderson, and A. Gurtov. RFC 3782 - The NewReno modification to TCP's fast recovery algorithm, Apr 2004. `ftp://ftp.rfc-editor.org/in-notes/rfc3782.txt`.

[13] R. Govindan and V. Paxson. Estimating router ICMP generation delays. In *Proceedings of Passive and Active Measurement (PAM'02)*, 2002.

[14] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary Internet end hosts. In *Proceedings of the 2nd ACM SIGCOMM Internet Measurement Workshop (IMW)*, Marseille, France, November 2002.

[15] V. Jacobson, R. Braden, and D. Borman. RFC 1323 - TCP extensions for high performance, May 1992. `http://www.faqs.org/rfcs/rfc1323.html`.

[16] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Inferring TCP connection characteristics through passive measurements. In *Proceedings of IEEE Infocom*, Hong Kong, March 2004.

[17] C. Jin, D. X. Wei, and S. H. Low. FAST TCP: Motivation, architecture, algorithms, performance. In *Proceedings of IEEE Infocom 2004*, Mar 2004.

[18] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of SIGCOMM'02*, Aug. 2002.

[19] T. Kelly. Scalable TCP: Improving performance in highspeed wide area networks. *ACM Computer Communication Review*, 33(2):83–91, Apr 2003.

[20] K. Lakshminarayanan and V. N. Padmanabhan. Some findings on the network performance of broadband hosts. In *Proceedings of the ACM/Usenix Internet Measurement Conference (IMC)*, Miami, FL, USA, October 2003.

[21] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet path diagnosis. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, USA, October 2003.

[22] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang. TCP Westwood: bandwidth estimation for enhanced transport over wireless links. In *Proceedings of MOBICOM'01*, pages 287–297, Jul 2001.

[23] M. Mathis and J. Mahdavi. Forward acknowledgment: Refining TCP congestion control. In *Proceedings of SIGCOMM 1996*, pages 281–291, Aug. 1996.

[24] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. RFC 2018 - TCP selective acknowledgment options, Oct. 1996. http://www.faqs.org/rfcs/rfc2018.html.

[25] A. Medina, M. Allman, and S. Floyd. Measuring the evolution of transport protocols in the Internet. *Computer Communication Review*, 35(2):37–52, 2005.

[26] A. Medina, M. Allman, and S. Floyd. Measuring interactions between transport protocols and middleboxes. In *Proceedings of the Internet Measurement Conference*, Taormina, Italy, August 2004.

[27] Monarch web site. http://monarch.mpi-sws.mpg.de/.

[28] M. Musuvathi and D. R. Engler. Model-checking large network protocol implementations. In *Proceedings of NSDI'04*, pages 155–168, March 2004.

[29] Netfilter: Firewalling, NAT, and packet mangling for Linux, 2006. http://www.netfilter.org.

[30] The network simulator – ns2. http://www.isi.edu/nsnam/ns/.

[31] J. Padhye. Papers on TCP modeling and related topics, 2001. http://research.microsoft.com/~padhye/tcp-model.html.

[32] J. Padhye and S. Floyd. Identifying the TCP behavior of web servers. In *Proceedings of the ACM SIGCOMM Conference*, San Diego, CA, USA, June 2001.

[33] V. Paxson. End-to-end routing behavior in the Internet. *IEEE/ACM Transactions on Networking*, 5(5):601–615, October 1997.

[34] V. Paxson, A. K. Adams, and M. Mathis. Experiences with NIMI. In *Proceedings of 2002 Symposium on Applications and the Internet (SAINT)*, Nara, Japan, February 2002.

[35] PlanetLab. http://www.planet-lab.org/.

[36] PlanetLab. Current slices in PlanetLab, 2006. http://www.planet-lab.org/php/slices.php.

[37] J. Postel. RFC 792 - Internet control message protocol, 1981. http://www.faqs.org/rfcs/rfc792.html.

[38] K. K. Ramakrishnan, S. Floyd, and D. L. Black. RFC 3168 - The addition of explicit congestion notification (ECN) to IP, Sep 2001. http://www.faqs.org/rfcs/rfc3168.html.

[39] R. Rejaie, M. Handley, and D. Estrin. RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the Internet. In *Proceedings of IEEE Infocom'99*, pages 1337–1345, Mar 1999.

[40] L. Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communications Review*, 1997.

[41] S. Saroiu, K. P. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 2002.

[42] S. Saroiu, K. P. Gummadi, and S. D. Gribble. SProbe: A fast tool for measuring bottleneck bandwidth in uncooperative environments, 2002. http://sprobe.cs.washington.edu.

[43] P. Sarolahti and A. Kuznetsov. Congestion control in Linux TCP. In *Proceedings of USENIX 2002*, June 2002.

[44] S. Savage. Sting: a TCP-based network measurement tool. In *Proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, USA, October 1999.

[45] University of Southern California, Information Sciences Institute. RFC 793 - Transmission control protocol, 1981. http://www.faqs.org/rfcs/rfc793.html.

[46] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, 2002.

[47] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: a mechanism for background transfers. In *Proceedings of OSDI'02*, Dec 2002.

[48] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[49] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control for fast long-distance networks. In *Proceedings of IEEE Infocom*, Hong Kong, March 2004.