Extracting Application Objects from TCP Packet Traces

Shane Alcock, Daniel Lawson and Richard Nelson
WAND Network Research Group
Department of Computer Science
University of Waikato
New Zealand
Email: salcock@cs.waikato.ac.nz, daniel@wand.net.nz, richardn@cs.waikato.ac.nz

***Abstract*—Network packet traces can be used to derive traffic models suitable for simulation or emulation of networks. These traces are shaped by the conditions on the original network the trace was taken from such as latency, loss, congestion and path MTU. Using a trace to derive a model can give plausible results, but the model will strongly reflect where the trace was collected. A different approach is to extract application objects and use them to derive traffic models free of network artefacts for simulation and emulation tasks. We propose a new algorithm for extracting application objects from a TCP packet trace by using non-MSS sized packets to demarcate the boundaries of objects. Unlike other object extraction methods, our approach does not require full packet capture; the TCP/IP headers are sufficient. We discuss the implementation of this algorithm and validate our algorithm against an existing object extractor that requires full payload capture.**

I. Introduction

Simulation is a powerful tool for studying complex computer networks such as the Internet. Network simulation is dependent on traffic models that describe how the simulated applications and agents should behave. Developing realistic traffic models is important to ensure that the results of simulations are accurate and applicable to the Internet.

Network packet traces encompass the variety of traffic seen on that network. This suggests that realistic traffic models can be derived directly from packet traces. However, as discussed by Floyd and Paxson [1], the packet-level behaviour is "shaped" by the characteristics of the network that the trace was captured from. Any traffic model based on packet-level behaviour will be similarly shaped.

Such traffic models have limited interaction with the simulated network and are unable to adjust correctly to network conditions. The timing of packets in the original trace will be preserved, even if the simulated network is faster or more congested. Packets that were retransmitted in the original trace will also be retransmitted in the simulation, even when the simulated packet was not lost. As a result, traffic models based on packet-level behaviour are not suitable for any simulation where the network conditions are different to the network the trace was captured from.

Packet-level traffic models are difficult to manipulate without adversely affecting the realistic nature of the model, particularly for a state-full protocol such as the Transmission Control Protocol (TCP) [2]. The properties and behaviour of a packet are dependent on other packets within the same connection. Duplicating, removing or displacing packets can disrupt the protocol interactions and produce unrealistic traffic patterns.

Instead we promote the use of *application objects* as a better means of generating traffic models from TCP packet traces. Application objects describe the source-level actions of software implementing an application-layer protocol. Through the use of application objects, it is possible to develop a traffic model that describes the behaviour of the applications rather than the transport protocols, eliminating the effects of packet-level behaviour while retaining the variety and realism of the original traffic.

A. Existing Work

Most current source-level traffic modelling approaches involve constructing models that describe application-layer protocols using application logfiles [3] [4], such as mail or web server logs, or full payload network traces [5] [6] as input. However, these models are specific to, at best, a small number of application-layer protocols whereas most modern networks feature traffic from a much larger variety of applications. Such models may be useful for analysing particular protocols but will not produce traffic that is indicative of the Internet as a whole. In addition, application logfiles and full packet captures typically contain data that is potentially sensitive. This can limit the diversity of input sources as some network operators, such as commercial Internet service providers, will be unable to disclose information without violating the privacy of their users.

An alternative approach has been to use NetFlow [7] sources as input to model derivation [8]. NetFlow provides flow summaries in a protocol independent fashion without prior protocol knowledge or access to sensitive payloads, eliminating the problems with the application specific traffic models. However, NetFlow summaries feature a high-level of abstraction. While this is helpful for modelling traffic across core networks, NetFlow summaries are not detailed enough to support work investigating TCP properties and other network effects as they lose a lot of the finer detail occurring at a packet level [9].

1-4244-1557-8/07/$25.00

An algorithm that can derive a model of the source-level behaviour from a packet header trace has been developed at the University of North Carolina [10]. The UNC algorithm does not require any specific knowledge of application-layer protocols and how they operate. Instead, object boundaries are identified by observing changes in the flow of packets. Although the underlying motivation is identical, there are several significant differences between the UNC algorithm and the technique we have developed. We discuss these differences in more detail in Section II-C.

B. Contribution of this Work

In this paper we present an algorithm that extracts application objects from packet header traces. Existing object extraction tools instead focus on identifying the protocol objects which describe the interactions within the application-layer protocol. Our algorithm is novel because the objects that are extracted are more representative of the behaviour of the applications involved.

Many contemporary object extraction methods are specific to a small number of application-layer protocols and require full payload captures to operate effectively. Our method uses only the information available in the TCP/IP headers to determine object boundaries and, as a result, can be applied to any application-layer protocol. Our algorithm can also successfully extract objects from traces where the packet payload has been encrypted or removed for privacy. The resulting output is a traffic model that encompasses all the application-layer protocols observed in the packet trace while retaining a high level of detail and, by using a packet trace as the traffic source, we can ensure that the traffic model will be based on real measured Internet traffic.

The remainder of the paper is structured as follows. In Section II we define application and protocol objects and discuss the differences between them. We also detail our technique for extracting application objects from a packet header trace. In Section III we describe the validation of our technique against an existing object extraction tool. Finally, we discuss the potential future work in Section IV.

II. Object Extraction

We define an *application object* to be a single data element that has been written to the network by a software application. Application objects describe the behaviour of the software implementations that utilise application-layer protocols. For example, consider a web browser implementation that uses two write system calls to transmit a HTTP GET request; the first contains the HTTP header, and the second contains the requested URL. Although this is only a single transaction within the HTTP protocol, the software is creating two separate application objects.

We define a *protocol object* to be a single data element that describes an application-layer protocol data unit (PDU). A protocol object is independent of the software that is transmitting the PDUs and may consist of multiple application objects, depending on how many system calls the application uses to write the protocol object to the network. The HTTP GET request described above would always be considered a single protocol object, no matter how many writes the application required to transmit it.

Both application and protocol objects are packetized as they are delivered across a network. It is these packets which are captured and written to disk to create network packet header traces. These packet traces preserve the packet-level behaviour of the traffic and are unsuitable for model derivation. We use the term *object trace* to refer to a time ordered sequence of records describing the TCP connections and their objects observed across a network. Unlike a packet header trace, an object trace describes only the source-level transactions and does not preserve any TCP artifacts that are present in the packet trace.

The TCP connections described within an object trace can be manipulated cleanly, as each connection appears as a separate entity within the trace. The operations discussed in [2], such as flow displacement, duplication or stretching, can all be applied when using application objects. The simulation engine in this case will deal with creating the lower level interactions, such as ARP requests, and any packet retransmissions required due to network effects.

Another benefit an object trace has compared to a packet trace is relative compression. An object may represent multiple packets but will only form a single record in an object trace. Only information pertinent to the source-level interactions needs to be recorded in the trace, resulting in smaller records. Much of the information stored in packet traces, such as TCP sequence and acknowledgement numbers, can be discarded without affecting a source-level traffic model.

Some application-layer protocols already feature support for obtaining object traces. Web server and web proxy logs can both provide a form of object trace, although the information they contain may not always be sufficiently detailed to make them useful for developing traffic models. For example, web server logs may only describe the size of HTTP objects downloaded by clients but not the size of the HTTP requests for those pages. Additionally, many Internet applications do not generate logs at all, limiting the range of application-layer protocols that can modelled in this manner.

Existing source-level traffic modelling techniques focus on identifying and analysing protocol objects. We believe this is not the best approach. The aim of using measured Internet traffic as a source for model derivation is to ensure the traffic model is as close to “real” traffic as possible. By using protocol objects, the idiosyncrasies of the individual software applications are lost, reducing the degree of realism present in the traffic model. A traffic model based on application objects instead will preserve the software behaviour and reflect the variety of application behaviour that is observed on the Internet more accurately.

A. Application Object Extraction

For the purposes of automatically extracting application objects from a TCP packet trace, we assume that applications

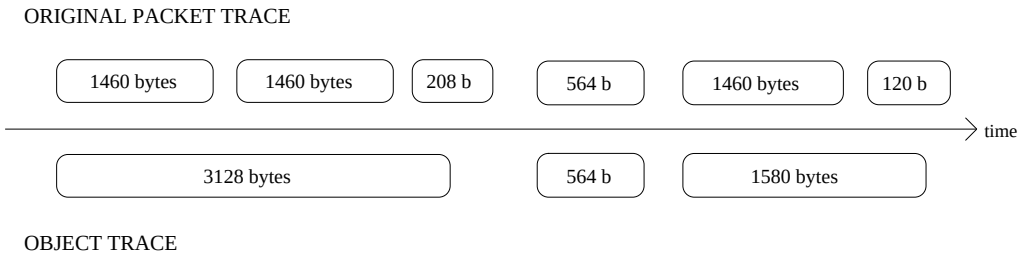

Fig. 1. A demonstration of the object detection algorithm. The packets are shown in sequence number order from left to right.

with large amounts of data to send will try to commit all of it to the network stack at once and the stack will create segments that are as large as possible. The maximum segment size (MSS) supported by a TCP connection is announced by each endpoint during the connection establishment process. We define an application object as *zero or more MSS sized packets travelling in the same direction, followed by a non-MSS sized packet*. An application object moves between connection endpoints in one direction only. There is no strict requirement on the direction of subsequent application objects within that connection; sometimes multiple unique objects are sent by one end of a connection without needing an intermediate object in the other direction.

Our algorithm reads each packet from a TCP packet trace and matches it to a half-connection based on the 5-tuple and the direction in which it was travelling. The 5-tuple consists of the server IP address, server port, client IP address, client port and the IP protocol field[1]. We define a half-connection as a series of packets travelling in a single direction as part of a TCP stream. A TCP connection between two hosts, A and B, consists of two half-connections: one made up of packets travelling from A to B and another consisting of packets travelling from B to A.

The packets observed for a half-connection are treated by our algorithm as a list, ordered by sequence number. Once the connection is over, either due to observing the connection tear-down handshake or expiring after a period of inactivity, the list of packets is traversed and the TCP payload size of each packet is compared to the MSS for that half-connection. Usually, the MSS for a half-connection is advertised on the initial SYN packet travelling in that direction. In cases where the MSS is not advertised or the advertised MSS is proven to be incorrect, we attempt to determine the MSS using a simple heuristic. The largest packet observed is checked to see if the total packet size, including TCP and IP headers, is either 576, 1420 or 1500 bytes in size. Each of these values is a common maximum transmission unit (MTU) encountered on the Internet. If a match is found, the size of the TCP payload for that packet will be used as the MSS. Otherwise, the MSS is regarded to be infinite, indicating that it was not possible to determine the MSS.

Packets with a payload size less than the MSS are designated to be the last packet in their respective object. However, the final packet seen for a half-connection is considered the final packet in the last object, even if it is MSS-sized. Thus, we can determine the boundaries between objects. Figure 1 illustrates the object extraction process; given an MSS of 1460 bytes, the third, fourth and sixth packets are determined to be object boundaries. Packets 1-3 form a single object of 3128 bytes, packet 4 forms an object by itself and a third object is formed by packets 5 and 6 that is 1580 bytes in size.

TCP control packets, such as the three-way connection handshake, are not considered as objects or as parts of objects and are used solely for determining connection information that aids the application object extraction process, e.g. expected sequence numbers, round trip time (RTT) estimates and the TCP state. Instead of being part of the traffic model, these packets will be created as required by the simulation or emulation environment.

One problem when performing application object extraction is the effect of Nagle's algorithm [11]. Nagle's algorithm states that if a user process commits less than a full segment to the network stack and there are still unacknowledged segments, to wait until either the unacknowledged data is fully acknowledged or until there is a full segment's worth of data. This could have the effect of combining two (or more) application objects together within the stack. This is a complete loss in resolution and, without protocol knowledge or packet payload inspection, cannot be resolved or detected by our method.

A further problem is that, on rare occasions, an application object will have been a multiple of the MSS in size. In such a case, the last packet seen for that object will be MSS-sized. Our algorithm will fail to detect that object boundary and the object will be combined with the subsequent object for that half-connection.

One feature of TCP that may have disambiguated such cases is the PSH flag in the TCP header. The PSH flag tells the receiver to transfer all buffered data to the application as the sender no longer had any data remaining in the send queue. Based on this, we expected the PSH flag to be set for every packet that appeared at the end of an application object. However, modern network stacks no longer use the PSH flag in this manner, often setting PSH for every packet they send.

[1]The IP addresses may have been anonymized by the packet capturing process. This will not affect the performance of our algorithm, provided the anonymization process does not translate two unique unanonymized IP addresses to the same anonymized IP address.

Because of this, attempting to use the PSH flag to identify application object boundaries greatly decreased the accuracy of our algorithm.

B. Performance and Implementation

Performance is a significant factor when dealing with network trace analysis [12]. The biggest constraint when performing object extraction is memory, as state information and packet records for all active connections is required to be kept. This is a particular problem for high bandwidth traces, as a large number of connections may be active at any point in time. As a result, we have implemented a slightly modified version of the algorithm presented in Section II-A that requires significantly less memory without affecting the end result.

To reduce memory consumption, our implementation maintains a 'current' object and the expected sequence number of the next packet for each half-connection. If an observed packet is the expected packet for that half-connection, it is immediately added to the current object and discarded. Packets that have sequence numbers greater than expected are buffered until the missing packet(s) arrive. To further decrease memory consumption, only pertinent packet information (such as the packet size and timestamp) is retained.

Packets with sequence numbers less than expected are retransmissions of packets we have already processed. If the retransmitted packet is larger than the original packet, it is possible the retransmit contains payload that has not been added into the current object. If this is the case, the current object size is increased to include the new payload. When the current object is deemed to be complete, i.e. a non-MSS sized packet is seen. the previous object is written to disk and the current object becomes the new previous object. At the conclusion of a connection, any objects still being stored locally are written to disk and the details of the connection itself are written to a separate file.

Once the object extraction process is complete, the connection and object output files are combined to produce an object trace. The finer details of the post-processing will not be discussed here, as it is primarily a sorting and categorising problem unrelated to the process of object extraction. Figure 2 demonstrates the trace format used by our implementation. Each record begins with details about the connection as a whole, such as the time the connection began and which ports were involved. The objects themselves appear at the end of the record in sequential order. The trace format presented is in plain text but it would be a simple task to convert to a binary format that would reduce the record size significantly.

In terms of actual performance of our implementation, a dual processor dual core 1.8 GHz Opteron machine can process a 24 hour packet trace (containing over 167 million packets) captured from the University of Waikato capture point in 25 minutes typically using 400-500 MB of RAM. This does not include the overhead of producing the final object trace from the two initial output files. A 4.2 GB compressed packet trace produces a 630 MB compressed object trace.

```
# Connection
server_ip=123.38.194.79
server_port=443
client_ip=52.57.188.150
client_port=1224
protocol=6
conn_rtt=0.301852
conn_starttime=1148515201.006697
# Direction Meta-data
dir=0 advert_mss=1380 observed_mss=1380 total_bytes=3075
dir=1 advert_mss=1460 observed_mss=1093 total_bytes=1261
# Objects
start=1148515201.434759 len=0.000000 dir=1 size=102 pkts=1
start=1148515201.435508 len=0.000000 dir=0 size=146 pkts=1
start=1148515201.723525 len=0.000000 dir=1 size=67 pkts=1
start=1148515202.095704 len=0.000000 dir=1 size=1092 pkts=1
start=1148515202.098881 len=0.000700 dir=0 size=2929 pkts=3
```

Fig. 2. An example connection from an object trace.

C. Comparison with UNC Method

The UNC method described in [10] determines object boundaries by observing a change in the direction of data transmission. The object traces produced, referred to as *a-b-t* traces, can be replayed using a tool called `tmix` [13]. The UNC method extracts the protocol objects rather than the application objects. The UNC method is reliant on the packet header trace being bidirectional to extract protocol objects. Our algorithm can produce an application object trace from both unidirectional and bidirectional packet traces. Although the protocol objects are identical to the application objects in many cases, a model derived from an application object trace will provide a better representation of the real-world traffic.

Some application-layer protocols may send multiple objects in one direction without requiring corresponding objects in the reverse direction. Those objects will be incorrectly combined into one single object by the UNC method, whereas our algorithm will detect that they are individual objects. The resulting traffic models may perform differently in simulation. Consider the example of an instant messaging client that sends three separate messages with a five minute gap between each message. The UNC traffic model would consider this a single object that took ten minutes to transmit. A simulation where the link bandwidth is doubled would see that object transmitted in five minutes. In reality, improving the bandwidth would have minimal impact on the instant messages as the delay was due to external factors unrelated to the network. This would be reflected in our traffic model which would describe three objects separated by significant idle time.

III. Validation

The object traces generated by our implementation are only useful if they accurately reflect the source-level behaviour of the applications captured by the original packet trace. To test the accuracy of our method, we can compare the object trace produced by our implementation against the output of existing object extraction methods that are more limited in their capabilities. One common limitation of other tools is that they can only reassemble objects from HTTP traffic due to their reliance on protocol-specific knowledge to perform object extraction. The full packet contents are also required.

Because of these limitations, we have been able to validate our algorithm's performance for the HTTP protocol only.

The tool that we used for the validation process was `ethereal`[2] [14]. `ethereal` is a protocol analysis tool that is capable of identifying HTTP objects by examining the value of the Content-Length field in the HTTP header which describes the size of the protocol object. To do this, `ethereal` requires the input packet trace to contain a significant quantity of packet payload, unlike our method which will only examine the packet headers. Initial testing concluded that `ethereal` did not feature any obvious bugs and that we would be able to conduct a comprehensive and meaningful validation experiment using it.

The packet trace we used through the validation process was a full payload capture from a single web server. The trace contained 10380 HTTP connections and over 630 thousand HTTP packets. The trace was bidirectional and was captured using the packet capture library `libtrace` [15] over the course of 44 hours and 20 minutes. Although the packet trace contained full payload, our implementation only examined the packet headers, i.e. up to and including the TCP header. The payload was only captured because it was required for `ethereal` to correctly classify HTTP objects.

When comparing the final results, HTTP connections that were regarded as incomplete, i.e. they did not both start and finish within the duration of the packet trace, were ignored to ensure the fairest possible validation scenario. For example, `ethereal` may be able to report the full size of an object that spans beyond the end of the trace if the HTTP Content-Length is available, whereas our implementation can only report the proportion of the object within the trace. We also ignored HTTP connections which did not include the Content-Length field in the HTTP header as it was impossible for `ethereal` to discern the application objects without that information.

For each connection we compared the sizes of the objects extracted by `ethereal` and our implementation. If the reported object sizes for both directions did not match perfectly, we manually examined the objects for that connection to determine the cause of the discrepancy. As `ethereal` reports the size of protocol objects and our implementation extracts application objects, it is important to determine whether any discrepancy was caused by behaviour that application objects should preserve rather than a failure of our algorithm. Occasionally, the variation in objects was caused by `ethereal` being unable to extract the HTTP objects correctly. As the primary purpose of `ethereal` is not object extraction, this was not an unexpected occurrence and such failures were noted in the final results.

A. Analysis

The results of the validation process are shown in Table I. 38 of the 10380 HTTP connections in the packet trace were incomplete and 633 further connections did not include Content-Length fields in the HTTP packets, leaving 9709 connections that could be used for validation. The application objects extracted by our implementation differed with `ethereal` on 240 of the connections - 2.47% of the total validation set.

TABLE I
ANALYSIS OF DISCREPANCIES ENCOUNTERED DURING VALIDATION

Reason	Frequency	% of Connections
Bad Advertised MSS	1	0.01%
Bad VLAN MSS	12	0.12%
Ethereal	12	0.12%
Object Size = MSS	4	0.04%
Mid-object Packet not MSS-sized	103	1.06%
Split HTTP GET	72	0.74%
Split HTTP POST	36	0.37%
Total	240	2.47%

Split HTTP GETs and POSTs account for 45% of the discrepancies observed when our implementation extracted application objects. This is caused by HTTP clients that split HTTP request objects, particularly GET and POST messages, into two separate parts. The first part contains the HTTP header and the second contains the message content. As the HTTP header is almost certain to be less than the MSS in size, our implementation will determine each request to be two separate objects whereas the Content-Length field of the HTTP header will describe it as a single object. This should not be regarded as a failure of our algorithm. A traffic model derived from our object trace would feature a small proportion of HTTP clients that split any GET or POST messages they transmit. A model based solely on the protocol behaviour, such as the object sizes reported by `ethereal`, would not contain any split HTTP requests and would therefore be less representative of real-world HTTP traffic.

Another significant cause of variation between the two programs is the observation of a non-MSS sized packet in the middle of an object, which caused our implementation to prematurely end the object. In most cases, we were not able to ascertain the exact reason a non-MSS sized packet was sent at that time. As a result, we have been unable to determine whether the discrepancy was caused by application behaviour, although we suspect this to be the case.

Several discrepancies were caused by incorrect configuration of the MTU along a Virtual Local Area Network (VLAN) path. The advertised MSS for that path was slightly larger than what the path could actually sustain, resulting in MSS sized packets being fragmented into one very large packet followed by a very small one. Both packets were determined to be distinct objects by our implementation. We believe that we can resolve this problem by updating our implementation to reassemble fragmented IP packets before considering them for object extraction.

Objects that were a multiple of the MSS in size was another potential flaw with our algorithm that we had noted earlier. There were only four instances of this scenario that caused an inconsistency in our validation, suggesting that this is a very minor problem that is unlikely to significantly affect any resulting traffic models. However, it is also possible that other

[2]ethereal has recently undergone a name change and it is now known as wireshark. ethereal version 0.10.10 was used for validation and was released prior to the name change.

application-layer protocols will transmit such objects more frequently. More validation would be required before we can be certain that the problem is insignificant.

Finally, a small but notable number of discrepancies were caused by `ethereal` being unable to extract protocol objects correctly. In some cases, an HTTP protocol object had some extra bytes of data prior to the HTTP header and `ethereal` was unable to locate the Content-Length field as a result. Another such situation was caused by the exact same connection 5-tuple appearing in the trace on two separate occasions. Our object extraction technique expires connections that have completed, but `ethereal` combined the two connections together. We aggregated all the discrepancies caused by `ethereal` into a single category in Table I.

At this stage we have not validated our implementation against other application protocols. We have been unable to locate any existing tools that are able to reliably extract objects for protocols other than HTTP to validate against. `ethereal` does not understand other common application-layer protocols to the same extent that it understands HTTP. Further validation will likely require the development of our own object reassembly tools that understand other common application protocols, such as SMTP.

IV. Future Work

The validation work presented in Section III showed that our method extracts application objects with a high level of accuracy, albeit only for some HTTP applications. We want to validate our implementation against other application-layer protocols. However, we have not yet discovered any tools that can accurately extract objects for protocols other than HTTP to compare our algorithm against. It is likely that we will have to develop our own software that can understand these protocols to perform more comprehensive validation.

The next logical step is to translate the object traces produced by our implementation into traffic models that can be used in simulation and emulation experiments. We have already begun work on a object replay system which will directly replay the connections and objects from the object trace, similar to `tmix` [13] which replays the *a-b-t* traces produced by the UNC object extraction method. We are also investigating using machine learning techniques to derive a Markov chain from the object traces produced by our implementation. The Markov chain can be used to stochastically generate synthetic traffic within a network simulation.

V. Summary

Evaluating network protocols and properties is typically done using simulation. For the results of those simulations to be applicable to the Internet, the traffic models used as input need to be as realistic as possible. Packet header traces provide a source of realistic Internet traffic but traffic models derived directly from packet traces preserve the packet-level behaviour specific to the conditions and characteristics of the source network. Such models can provide misleading results when used as input to a simulation experiment.

Instead, we propose the use of application objects for derivation of models suitable for simulation engines. Application objects capture the variety of application-layer protocols and their implementations active on the modern Internet better than the protocol objects used by existing source-level traffic modelling tools. We have developed an algorithm that can extract application objects directly from a TCP packet header trace that uses non-MSS sized packets to determine the object boundaries. The algorithm can extract objects for any application-layer protocol present in the trace and requires no protocol-specific knowledge to do so. The resulting object traces retain the realistic nature of the measured traffic but the TCP artefacts and packet-level behaviour are eliminated.

We have written an implementation of our algorithm and validated it against a protocol object extraction mechanism that relied on both application-specific knowledge and a full payload packet trace, which proved that our implementation was able to extract application objects with a high degree of accuracy despite the comparatively limited information available to it.

As a result, we believe that the work presented in this paper has the potential to significantly improve the quality of network simulation and emulation experiments.

References

[1] S. Floyd and V. Paxson, "Difficulties in simulating the internet," *IEEE/ACM Transactions on Networking*, February 2001.

[2] A. Rupp, H. Dreger, A. Feldmann, and R. Sommer, "Packet trace manipulation framework for test labs," in *IMC '04: Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement, (New York, NY, USA, 2004)*, pp. 251–256, 2004.

[3] A. J. McGregor, M. W. Pearson, and D. H. T. R. Lawson, "Validation of the wand simulator through comparison with laboratory tests," in *Proceedings of the PAM2002 Workshop on Passive and Active Measurements, (Ft Collins, Colorado, March 2002)*, 2002.

[4] M. W. Pearson and A. J. McGregor, "Sensitivity analysis of event driven simulation results," in *Proceedings of the PAM2001 Workshop on Passive and Active Measurement, (Amsterdam, April 2001)*, 2001.

[5] J. Hall, I. Pratt, and I. Leslie, "Non-intrusive estimation of web server delays." http://www.cl.cam.ac.uk/research/srg/netos/netx/publications/sdel.html, 2001.

[6] B. A. Mah, P. Sholander, L. Martinez, and L. Toldendino, "Ipb: An internet protocol benchmark using simulated traffic," in *MASCOTS '98: Proceedings of the 6th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, (Washington, DC, USA, 1998)*, p. 77, 1998.

[7] "Cisco netflows." http://www.cisco.com/warp/public/732/netflow.

[8] C. Barakat, P. Thiran, G. Iannaccone, C. Dior, and P. Owezarski, "A flow-based model for internet backbone traffic," 2002.

[9] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, and M. Khan, "Packet-level traffic measurements from the sprint ip backbone," 2003.

[10] F. Hernández-Campos, F. D. Smith, and K. Jeffay, "Generating realistic tcp workloads," in *Proceedings of CMG2004 Conference*, pp. 273–284, 2004.

[11] J. Nagle, "Congestion control in ip/tcp internetworks, rfc896," tech. rep., IETF, 1984.

[12] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, J. Ioannidis, M. B. Greenwald, and J. M. Smith, "Efficient packet monitoring for network management," in *Proceedings of IFIP/IEEE Network Operations and Management Symposium (NOMS) 2002*, 2002.

[13] M. C. Weigle, P. Adurthi, F. Hernández-Campos, K. Jeffay, and F. D. Smith, "Tmix: A tool for generating realistic application workloads in ns-2," *ACM SIGCOMM Computer Communication Review (CCR)*, vol. 36, no. 3, pp. 67–76, 2006.

[14] "Ethereal: A network protocol analyser." http://www.ethereal.com/.

[15] "Libtrace." http://research.wand.net.nz/software/libtrace.php.