Paisley, J. and Sventek, J. (2006) Real-time detection of grid bulk transfer traffic. In, *The Tenth IEEE/IFIP Network Operations and Management Symposium 2006 (NOMS 2006), April 2006,* pages pp. 66-72, Vancouver, British Columbia, Canada.

Real-time Detection of Grid Bulk Transfer Traffic

Jonathan Paisley and Joseph Sventek
Department of Computing Science
University of Glasgow
Glasgow, United Kingdom
Email: {jp,joe}@dcs.gla.ac.uk

Abstract— **The current practice of physical science research has yielded a continuously growing demand for interconnection network bandwidth to support the sharing of large datasets. Academic research networks and internet service providers have provisioned their networks to handle this type of load, which generates prolonged, high-volume traffic between nodes on the network. Maintenance of QoS for all network users demands that the onset of these (Grid bulk) transfers be detected to enable them to be reengineered through resources specifically provisioned to handle this type of traffic. This paper describes a real-time detector that operates at full-line-rate on Gb/s links, operates at high connection rates, and can track the use of ephemeral or non-standard ports.**

I. Introduction

The practice of scientific research is increasingly associated with the extraction of information from exponentially increasing volumes of experimental data [1]; examples abound in bioinformatics, geophysics, astronomy, medicine, engineering, meteorology and particle physics. Ever-larger processing and communication resources are required to support such information extraction. Significant financial support is provided by a number of governmental organizations (e.g., UK e-Science [2], EGEE [3], TeraGrid [4]) to facilitate this practice.

The term 'Grid computing' embodies the idea of dynamically-managed, wide-area distributed computing. One definition of a Grid is: "... a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed autonomous resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements." [5]

Information extraction typically requires analysis and correlation of multiple datasets. Once processing resources have been chosen, the requisite data must be made available to these processing resources. While there are research activities in support of distributed queries to remote data [6], much of the current practice consists of transfers of entire experimental data collections for processing by a local cluster of processors. As the datasets in many experimental domains are extremely large, such transfers can consume a considerable portion of the bandwidth available from academic research and/or commercial networks. The recent developments of a number of high-speed TCP variants [7], [8] can only exacerbate the situation.

Of primary concern to operators of academic research and/or commercial networks is the impact that prolonged, high-volume traffic, such as transfer of datasets, has on the quality of service perceived by other users of their networks. Most such operators will have provisioned their networks sufficiently to handle the growing bulk transfer traffic (BTT), often by explicitly provisioning routers and links to carry this type of traffic. The extent to which other traffic is affected (such as interactive web browsing, terminal sessions or simple email) will depend on the provisioning and the accuracy of routing decisions. We aim to provide operators with additional information to better manage provisioning and routing.

If they are able to detect the onset of bulk transfer activity, they can reengineer the bulk traffic onto specific resources using MPLS [9] or other techniques. For example, our expectation is that operators will use MPLS to define multiple paths between ingress and egress routers. Therefore, the primary use of our events would be to add the newly found quadruple to the FEC definition in the ingress router, thus mapping to a path specifically provisioned for Grid bulk data transfer traffic. Due to the dynamic nature of Grid applications, statically configured routing table entries are unlikely to effectively manage the Grid network traffic.

The hypothesis of this work is that it is possible to detect the onset of bulk transfer activity in real-time, using a variety of techniques. This paper discusses a detector based upon one such technique.

II. Related Work

A. Passive Application Identification

Currently, there are three widely used approaches for passive application traffic identification: application signatures, transport layer ports and network/transport layer application pattern recognition based on heuristics.

The application signature approach [10] searches for application protocol specific patterns inside packet payloads. While simple to understand, it introduces some significant problems. First, it cannot be adapted automatically to unknown, recently introduced application protocols since the protocol state machine for each application of interest must be known. Second, application-level pattern search in transport packets, usually achieved by reconstruction of individual flows, generates significant processing load; application of such systems to higher-speed network links (1 Gb/s and higher) usually results in overload for the software, resulting in dropped packets. Finally, some application protocols avoid payload inspection by using encryption algorithms.

1-4244-0143-7/06/$20.00 ©2006 IEEE.

Transport layer port identification [11] addresses the load and encryption problems, as it does not produce much load at the measurement nodes and does not rely on inspecting application payloads. This method still suffers from inability to adapt to modified or recently introduced protocols. Furthermore, many applications have begun using ephemeral port numbers to deliberately avoid port-based identification. As a result, port-based application identification can highly underestimate the actual application traffic volume [12].

Heuristic based network/transport layer approaches [12], [13] use simple network/transport layer patterns, e.g., the simultaneous usage of UDP and TCP ports and the packet size distribution of an application flow between components of the application. This method can give good performance for existing application protocols and may even be used to discover unknown protocols. Two problems exist with this approach: it may be straightforward to construct a new application protocol that avoids any particular heuristic, and it may be difficult to eliminate false positives.

B. Intrusion Detection Systems

There has been substantial research interest in intrusion detection systems [14]. These systems attempt to classify the traffic seen passively at a monitoring point, and raise an alarm if a potential intrusion has been detected. Such an intrusion detector acts in a similar fashion to the bulk transfer traffic detector—instead of looking for potential security intrusions, the BTT will raise an alarm when bulk transfers are detected.

Bro [15] is one such IDS that is well known in the community. Bro reassembles each individual TCP flow, and provides for different patterns to be matched against these reconstructed flows. Bro uses libpcap [16] to statically install packet filters to yield the packets that are reassembled into flows. Bro suffers from two drawbacks for use in the bulk transfer monitoring domain:

- the packet filters are statically specified; if one is attempting to track the use of ephemeral ports, then one must be able to modify the filters on the fly;
- as constructed, Bro cannot handle full line rate traffic at Gb/s speeds.

C. Grid Bulk Transfer Applications

As described in Section I, the primary goal is to detect the onset of Grid bulk transfer traffic that can affect the QoS delivered to other users of the network. Most Grid bulk transfer traffic is generated by a bounded set of applications—e.g., Storage Resource Broker (SRB) [17], GridFTP [18] and bbFTP [19].

Bulk data transfer application protocols define two types of data flow:

- control traffic that determines the data to be transferred and various characteristics of the actual data transfer;
- data traffic to actually transfer the data between the participants.

These logical sub-flows have very different characteristics; the control sub-flow exhibits the usual characteristics of a short-lived, remote procedure call protocol, while the data sub-flow exhibits the characteristics of a 1-way, bulk flow. It is the latter that are expected to most affect the QoS perceived by other users of the network.

Most transfer applications separate the control and data sub-flows into separate TCP flows (e.g., GridFTP or bbFTP); others mix the control and data sub-flows over a single TCP flow (e.g., SRB). Well-known ports are often used for establishing the control flow, while the data flow takes place on a TCP connection using ephemeral ports. However, site-specific firewall restrictions may cause local reassignment of the established port numbers, meaning that they are no longer a reliable indicator of protocol/application identity.

The protocols that mix control and data sub-flows over a single TCP flow provide a challenge to a detector attempting to operate at Gb/s line rates, as the detector may spend a significant percentage of its cycles processing the bulk data traffic; once the bulk data flow has started, the detector will not learn much from the bulk data, but it must be prepared to process the next control message that follows the current bulk transfer.

D. Summary

The initial focus of the work described herein has been to construct a detector that uses the application signature approach to address two specific issues discussed above: the ability to handle full-line-rate traffic at Gb/s speeds and the ability to track the use of ephemeral ports.

Reassembly of all TCP flows in real-time stresses the engineering of the detector if one is to avoid packet loss; additionally, given the predominance of mice over elephants [20] in terms of proportion of Internet flows, the detector will be stressed by sudden bursts of flow creation.

III. The Grid BTT Detector

This section describes the requirements, assumptions, design characteristics, and implementation features necessary to meet the goals for the detector.

A. Requirements

It is essential that the detector be able to operate at full Gigabit Ethernet line rate. In particular, it must handle large numbers of concurrent TCP flows and full line rate in both directions.

The detector must support simultaneous analysis of a wide range of bulk transfer protocols without resorting to well-known port identification of control streams. This requirement implies that before a protocol analyzer is attached to a particular TCP flow, an application identification module has the opportunity to analyze the initial content of every flow in order to classify it according to the current set of installed protocol analyzers.

Since the protocols under analysis are port-agile, it must be possible to adjust packet filters with causal effect following reception of a particular packet that causes the filter to be updated.

1-4244-0143-7/06/$20.00 ©2006 IEEE.

A complex event reporting mechanism is NOT required. The detector delivers low-level events to an unspecified management system that can process the events as required. The output from the detector is a set of event messages describing application type, any associated flow identifier tuples, and any additional information available from the protocol analysis (such as size of file being transferred). Conversion of this raw data into a form suitable for processing by a management system should not be too difficult.

For example, we could define or reuse a MIB that contains such information, and for which an interested management application could register to receive an SNMP trap whenever a new data flow has been identified, or whenever a data flow associated with a different source/destination quadruple has been detected.

We expect network operators typically to use these detection events to facilitate online traffic re-engineering of Grid bulk data across, for example, specially provisioned paths.

B. Assumptions

- TCP only—it is assumed that all applications of interest are using TCP.
- IPv4 vs. IPv6—both IPv4 and IPv6 are supported.
- No IP fragment handling—properly configured end hosts should be able to negotiate the appropriate maximum segment size for the end-to-end path. If evidence later suggests that IP fragments are more prevalent than assumed here, then additional reassembly support could be added.
- Studied applications are not used maliciously.

C. Design

In order to meet the 'full-line-rate in both directions' requirement, as well as to be able to causally modify the filtering criteria to handle ephemeral ports, the most important design characteristic is to choose appropriate, affordable, monitoring hardware that will enable a user-level application to handle all packets without interrupting the processor. The chosen monitoring hardware was a GIGEMON system [21], which consists of a DAG4, GigE dual channel network monitoring card and a 2.8 GHz dual Xeon processor with 2+ GB of memory. A passive optical tap delivers copies of the packets on the GigE substrate for processing by the DAG card; the DAG card makes the packets available via a ring buffer in memory over which the monitoring application is mapped.

Given this memory structure, the detector design has adhered to the following design principles:

- Minimize memory copies—packet data should be copied as little as possible to minimize CPU load and to reduce contention on the PCI bus (shared by the DAG card the and CPU). In particular, if the bus is busy, the DAG may drop packets.
- Minimize heap allocation—heap-based memory allocation/deallocation can be time consuming and causes heap fragmentation. This is a particular concern for a long-running system such as the detector.

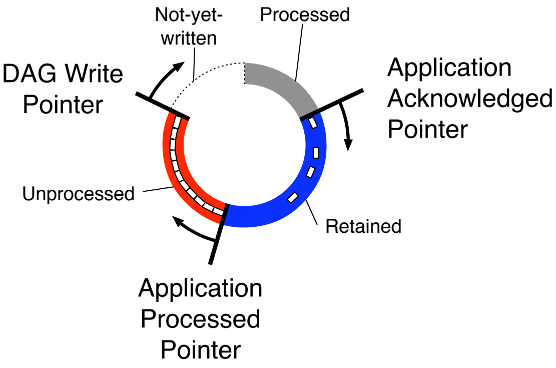
Not-yet-written
Processed
DAG Write Pointer
Application Acknowledged Pointer
Unprocessed
Retained
Application Processed Pointer

Fig. 1. DAG ring buffer.

- Process packets as soon as possible—if packets are not processed when they arrive, they must be buffered. Buffering costs time and memory, and should be avoided where possible.
- Single-threaded, data-driven structure—to simplify implementation and eliminate synchronization issues, the system is to run on a single processor, with processing driven by the arrival of packets.

1) Main Event Loop: The detector is structured around a packet-driven main loop. The DAG card stores incoming packets into a traditional circular buffer of several hundred megabytes. The buffer, with some additional annotations that are explained later, is shown in Fig. 1.

Packets in the buffer are processed by the detector application; the DAG card stores further packets into the buffer in parallel. Any non-IP packets are immediately skipped, and subsequently any non-TCP packets are also skipped. A canonical quadruple consisting of [source port, source address, destination port, destination address] is formed for use as a key into the TCP connection table. The absence of an entry in the TCP connection table causes a new connection record to be allocated and installed in the table. The packet is then dispatched to the handler associated with the connection.

The main loop handles one other type of event: timers. These are set up by the connection or protocol handlers, and are primarily used to eliminate connection records after inactivity or once a connection has been closed. The timer queue is processed periodically but not as often as once per packet at times of high load. The current implementation avoids locking overheads (both at run time and in the extra consideration required for multithreaded programming) by dispatching everything from a single thread. As such, the second processor is not used.

We intend to evaluate the tradeoffs between this design and alternatives such as managing timers on a second processor.

2) Connection Processing: Each connection record is composed of two flow records, one for each direction of the TCP flow. The flows track the state of the TCP stream and deliver incoming segments to the protocol analyzer when they arrive.

Out of order segments are treated specially—they are left in

1-4244-0143-7/06/$20.00 ©2006 IEEE.

the DAG capture buffer, and added to a per-flow linked list of pending packets; these packets are termed ‘retained packets’. Additionally, a global linked list is constructed to enable the main event loop to keep track of the lower limit in the circular buffer (see Fig. 1). Once a packet arrives representing the next segment to be delivered to the application, any retained packets are processed and delivered, if appropriate.

If the main event loop determines that available space in the circular buffer is limited (due to the DAG write pointer nearing the earliest retained packet), the owning flow is given the opportunity to spill the packet out of the circular buffer onto the heap. The threshold for performing this spilling is currently half the capture buffer size.

This threshold gives a large safety zone for incoming data captured by the DAG card to be stored. Since packet processing is incrementally data-driven, the amount of time taken to process each packet is bounded, as protocol analyzers do not block; as such, retained packets can be spilled incrementally to keep the ring buffer sufficiently empty. One is, however, trading off storage safety for incoming packets against the costs to spill retained packets to the heap (a heap allocation and a copy).

By considering a detector monitoring a Gigabit Ethernet link, basic figures for the duration of time a packet could remain in the buffer can be determined. Given a 1 GB ring buffer and maximum of 2 Gb/s fill rate (full line rate in both directions), the buffer will be filled in around 4 s. Therefore, a retained packet (assuming the current fill threshold of 50%) would have a minimum lifetime of 2 s, or proportionally longer for links with lower data rates.

3) Application Identification: Before a TCP connection has a protocol analyzer attached to it, it is necessary to identify the correct protocol. If a connection cannot be identified within the first few packets or limited number of bytes, it is abandoned.

Protocol identification in the general case can be implemented by regular expression string matching on packet contents [22]. For some applications, this simplifies to performing a simple string comparison with the packet contents—e.g., SRB with its signature ‘`START SRB\0`’ string.

The net result of application identification is that the appropriate protocol analyzer is associated with each flow.

4) Protocol Analyzer Interface: In-order data segments are delivered to the appropriate protocol analyzer. The analyzer, in turn, informs the reassembler of the amount of data it is expecting and whether it is interested in that data. This enables, for example, an RPC-based transport protocol that carries both control and data over the same TCP flow (e.g., SRB in serial transport mode) to be monitored efficiently during the control traffic phase and during the bulk-data phase. The reassembler is informed that a particular range of bytes are uninteresting, and therefore matching packets do not need to be delivered or buffered. Thus, in most cases of high-bandwidth packet reordering, packets do not need to be retained at all.

A further optimization is to compare new packets against existing retained packets in a flow. Any existing retained packets are, by definition, from not-ready-to-be-delivered future sequence numbers. Therefore, if a new packet arrives that is a retransmission of an existing retained packet, the new (later) packet can be substituted for the retained (earlier) packet; thus the DAG write limit pointer may be advanced as a result of freeing the retained packet.

```
void AnalyserClass::AnalyserMain()
{
    // Read function id and num args
    READ(OrigFlow, 2);
    func_id = *(uint16_t *)data;
    READ(OrigFlow, 2);
    num_args = *(uint16_t *)data;

    for (i=0; i<num_args; i++) {
        READ(OrigFlow, 4);
        len = *(uint32_t *)data;
        // Read the argument, but we
        // only need first 200 bytes
        READ_AND_SKIP(OrigFlow, len, 200);
        // ... process the argument
    }
    READ(RespFlow, 4);
    result_value = *(uint32_t *)data;
}
```

Fig. 2. Example of ProtoThread-based protocol analyser.

5) Protocol Analyzer Structure: A protocol analyzer implementation usually involves constructing an event-driven state machine (EDSM), where the events represent the arrival of in-sequence data packets from each direction of the connection, and the states correspond to the request-reply sequences of the protocol in question. This kind of code is difficult to understand and maintain, and can quickly become unwieldy.

To improve the maintainability of analyzers constructed for the detector, and consistent with the chosen single-threaded approach, a system known as ProtoThreads[1] [23], [24] is used. This system is a C language construct that enables straightforward imperative programming to describe the control flow, while being based on an EDSM model. No native platform threads are involved, and synchronization and locking do not have to be considered. An example of a ProtoThread-based protocol analyzer is given in Fig. 2.

D. Implementation

The system has been implemented in C++, providing the power and speed of C with the flexibility of object-oriented design. An earlier attempt to implement an SRB analyzer in Bro was written in C++; as a result it was straightforward to adapt it to work in this new environment.

Access to the DAG circular buffer is attained by linking against the Endace-supplied libdag. A libpcap-based version has also been implemented to provide a trace-driven testing interface.

The TCP connection table is a simple hash table; the timer manager is implemented using calendar queues.

[1] Proto because they are small and not-quite real threads. They achieve thread switching by unwinding the stack rather than traditional context switching, via a low-level C local continuations library.

Per-packet overheads are minimized by avoiding any heap allocation in the fast path. Protocol analyzers work on the data directly in the circular buffer (unless a packet has been spilled) so memory copies are avoided.

1) ProtoThreads: ProtoThreads are conceptually based upon continuations—i.e., the ability to save the state of an execution path as a continuation, proceed to execute other code, and resume from the continuation at a later time. ProtoThreads are a particularly light-weight implementation of this technique that is based on a specialisation called 'local continuations'. In this specialisation, rather than supporting resumption of code from arbitrary points in the program, it is only possible to resume execution from within the function that saved the continuation.

The code in Fig. 2 parses a series of length-prefixed arguments to an RPC call. A 16-bit function ID is sent, followed by the number of arguments (as a 16-bit integer). Each argument is specified as a 32-bit length followed by the appropriate number of bytes of data, with no padding.

The `READ` and `READ_AND_SKIP` calls are C pre-processor macros that check if there is sufficient data available to satisfy the request (by interrogating instance variables of the analyzer object that describe the data available in the current packet). If there is not enough data, the function returns after updating the relevant flow record to indicate how much data is required. The next time the function is called, execution resumes from the last `READ` or `READ_AND_SKIP` statement.

Since the analyzer function is invoked multiple times, local variables are not preserved across `READ` and `READ_AND_SKIP` statements. By using C++ as the implementation language, persistent state can be stored in instance variables in the analyzer class without adding any extra syntax.

2) SRB Protocol Analyzer: As an example, an SRB analyzer has been implemented that is able to analyze the basic RPC system that underlies the SRB protocols. There are many functions corresponding to traditional file system APIs such as open, close, read, write, seek, and stat, and additional functions that set up third-party transfers. The arguments or return values to these functions specify IP addresses and port numbers that will be associated with bulk data transport. As an additional level of complexity, different functions are used depending upon whether the connection is client-server or server-server.

IV. Validation of the Prototype

Port identification, ephemeral or otherwise, is a by-product of a detector based upon the application signature approach and full flow reassembly. Each hand-built protocol analyzer is able to track the use of ephemeral ports in the operation of the protocol. Detection of non-standard port use is provided by the application identification aspect of the detector. Note that the protocol analyzers depend upon strict adherence by the end applications to the defined protocol; as such, a detector is brittle with respect to ad hoc modifications to the application protocols by cooperating end systems.

For the types of applications under study (bulk data transfer) the ideal network utilisation scenario is when the contents of the files being transferred consume all available bandwidth. Although control protocol overheads make this unattainable, it is still expected that most link utilisation is accounted for by bulk data. In order to evaluate the ability of the prototype to handle full line-rate traffic on Gb/s links, we consider two extremes: one is when the link is saturated by bulk data traffic, the other when there is an unusually high amount of control-traffic corresponding to the applications for which we have protocol analyser modules.

Given a load of approximately 900 Mb/s in each direction with maximum-size packets, the CPU usage on the monitoring hardware is less than 1%. Under a trace-driven synthetic load of 4,000 concurrent SRB sessions resulting in 135 MB/s and 140,000 packets/s of control traffic, the peak CPU usage was 27%. Bulk data was eliminated from the traces in order to explicitly measure the performance of the protocol analyser module. This kind of network load would be a highly unusual scenario on a production network, but here shows that the performance of the prototype is sufficient to cope even with extremes of control/bulk data mix.

The SRB analyser in use could be further optimised to take better advantage of the retained packet scheme provided by the TCP stream reassembler. As currently implemented, RPC-style arguments used in the SRB analyser are copied onto the heap (incurring memory allocation and copying overhead). These arguments could generally by left in the packet buffer and accessed directly from there once the complete RPC call has been parsed.

The above metrics cover the performance of the monitoring system when dealing with traffic for the bulk transfer applications under study. Since other application traffic is likely to be present on the link, we also consider the behaviour when 'ignored' TCP flows are involved. Once a flow has been marked as uninteresting, the fast path just involves a hash table lookup, after inspection of the packet header. The main overhead in processing such connections is the initial allocation and hash table insertion of the connection control record and associated timer queue updates. We generated TCP SYN packets at a range of new-connection-rates to evaluate system load under extreme conditions. The results are shown in Fig. 3, with connection rates from 100,000 to 400,000 connections per second in steps of 20,000/s. Total CPU load (user and system time) scales linearly with the increase in connection rate, and the average sustained load for the 400,000/s rate is around 86% (with approximately 250 Mb/s bandwidth used). The rise at around 17 s corresponds to the first connection timers expiring to delete the connection records corresponding to the packets around 0 s. The noise after 90 s, when the load is switched off, is an artifact of timer processing when there are no regular incoming packets.

400,000 new connections per second is exceptionally high, and would be unlikely in a link which is mainly carrying bulk traffic. However, it is important to understand the limitations of the system. Other than CPU load, memory usage must also be considered. Each active connection control block consumes approximately 190 bytes and with an initial connection

1-4244-0143-7/06/$20.00 ©2006 IEEE.

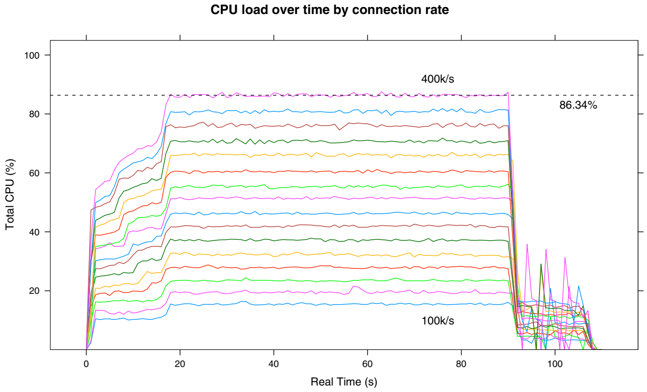

Fig. 3. CPU load over time for different connection rates.

timeout of 15 seconds (as implemented) the 6 million active connections consume over 1GB of memory. If the system starts using swap space, it can generally no longer keep up with real-time processing of incoming packets. Memory usage can be reduced by using mini-connection records for yet-to-be-established flows (which would be useful for dealing with denial of service attacks), and also by reducing the initial connection timeout.

Protocol analysers also incur a time and memory cost, but we expect their numbers to be significantly fewer than the total number of active TCP connections since the classification (see III-C.3) process discards unknown or unwanted connections before an analyser is instantiated. The analysers have been designed to use a small, bounded amount of memory during execution, which further limits resource consumption.

Each analyser (we have simple analysers for plain FTP, HTTP, SRB, bbftp and iperf) processes the control traffic on a connection. The time spent in the analyser is roughly proportional to the amount of data on the connection, but can be less in cases where parts of the control connection are known to be irrelevant and can be skipped. Each response body of HTTP/1.1 connections containing multiple objects can be ignored, for example.

The costs in scaling the number of different analysers (as opposed to the number of analysers operating concurrently) is attributed to the initial classification of flows. This is handled efficiently by a state machine against the initial flow data.

V. Summary and Future Work

The design for a real-time detector for the onset of Grid bulk transfer traffic has been described. A prototype implementation of the detector has been constructed using commodity PC hardware and readily available packet capture hardware. The prototype successfully tracks the use of ephemeral or non-standard ports by Grid bulk transfer applications, operates over full-line-rate Gb/s Ethernet links, and continues to function at high connection rates.

This represents just one part of a larger effort to construct on-line, real-time classifiers/detectors for applications of interest to Internet service providers [25]. Current efforts include study of how best to integrate this style of detector with detectors based upon heuristic pattern matching.

Most Grid bulk transfer protocols permit the control and/or data sub-flows to be encrypted, thus nullifying the ability for detectors based upon the application signature approach to discover bulk flows. Initial efforts are under way to construct a heuristic pattern detector that can operate on encrypted flows. By reasoning about application-level message exchanges (timing, size, direction) we aim to do real-time protocol classification and infer application-level state changes.

No attempt has yet been made to integrate the current detector with an existing Operational Support System that can reengineer the detected traffic through different network resources. Such validation is critical to show that these types of real-time detectors can positively affect the operation of a production network.

Acknowledgment

The authors wish to acknowledge partial support for this work by the UK Engineering and Physical Sciences Research Council under grant GR/S68989/01.

The authors would like to thank the members of the P2POpt project for the use of their GIGEMON systems. J.P. is particularly indebted to L. Mathy and D. Pezaros at Lancaster University for the use of one of their machines to conduct some of the performance experiments.

References

[1] R. Mann, R. Williams, M. Atkinson, K. Brodie, A. Storkey, and C. Williams, "Scientific data mining, integration and visualisation," National e-Science Centre, Tech. Rep. UKeS-2002-06, Nov 2002.
[2] About the UK e-Science Programme. [Online]. Available: http://www.rcuk.ac.uk/escience/
[3] Enabling grids for e-science. [Online]. Available: http://public.eu-egee.org/
[4] Teragrid. [Online]. Available: http://www.teragrid.org/
[5] R. Buyya. Grid Computing Info Centre: Frequently Asked Questions (FAQ). [Online]. Available: http://www.cs.mu.oz.au/~raj/GridInfoware/
[6] A. Anjomshoaa, M. Antonioletti, M. Atkinson, R. Baxter, A. Borley, N. Hong, B. Collins, N. Hardman, G. Hicken, A. Hume, A. Knox, M. Jackson, A. Krause, S. Laws, J. Magowan, C. Palansuriya, N. Paton, D. Pearson, T. Sugden, P. Watson, and M. Westhead, "The Design and Implementation of Grid Database Services in OGSA-DAI," in *Proc. UK e-Science All Hands Meeting*, Nottingham, UK, September 2003.
[7] C. Jin, D. Wei, and S. Low, "FAST TCP: Motivation, Architecture, Algorithms, Performance," in *Proc. IEEE Infocom*, Miami, FL, March 2004.
[8] R. Shorten and D. Leith, "H-TCP: TCP for high-speed and long-distance networks," in *Proc. PFLDnet*, Geneva, CH, February 2003.
[9] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol label switching architecture," RFC 3031, Internet Engineering Task Force, 2001. [Online]. Available: http://rtg.ietf.org/rfc/rfc3031.txt
[10] S. Sen, O. Spatscheck, and D. Wang, "Accurate, scalable in-network identification of p2p traffic using application signatures," in *Proc. 13th international conference on World Wide Web*, 2004.
[11] S. Sen and J. Wong, "Analyzing peer-to-peer traffic across large networks," in *Second Annual ACM Internet Measurement Workshop*, 2002.
[12] T. Karagiannis, A. Broido, N. Brownlee, K. Claffy, and M. Faloutsos, "File-sharing in the internet: A characterization of p2p traffic in the backbone," UCR, Tech. Rep., November 2003.
[13] T. Karagiannis, A. Broido, M. Faloutsos, and K. Claffy, "Transport layer identification of p2p traffic," in *Proc. ACM SIGCOMM conference on Internet measurement*, 2004.
[14] B. Mukherjee, L. Heberlein, *et al.*, "Network intrusion detection," *IEEE Network*, pp. 22–41, May/June 1994.

[15] V. Paxson, “Bro: A system for detecting network intruders in real-time,” *Computer Networks*, vol. 31, pp. 2435–2463, 1999.

[16] Lawrence Berkeley National Labs Network Research Group. libpcap. [Online]. Available: http://ftp.ee.lbl.gov

[17] C. Baru, R. Moore, A. Rajasekar, and M. Wan, “The sdsc storage resource broker,” in *Proc. CASCON’98*, Toronto, Canada, 1998.

[18] W. Allcock, A. Chervenak, I. Foster, C. Kesselman, and S. Tuecke, “Protocols and services for distributed data-intensive science,” in *Proc. Advanced Computing and Analysis Techniques in Physics Research (ACAT)*, 2000, pp. 161–163.

[19] bbftp - large files transfer protocol. [Online]. Available: http://doc.in2p3.fr/bbftp/

[20] K. Thompson, G. Miller, and R. Wilder, “Wide-area internet traffic patterns and characteristics,” *IEEE Network*, vol. 11, no. 6, 1997.

[21] Endace Measurement Systems—Server Based Solutions—GIGEMON. [Online]. Available: http://www.endace.com/gigemon.htm

[22] R. Sommer and V. Paxson, “Enhancing byte-level network intrusion detection signatures with context,” in *Proc. ACM CCS*, 2003.

[23] A. Dunkels, O. Schmidt, and T. Voigt, “Using protothreads for sensor node programming,” in *Proc. REALWSN’05 Workshop on Real-World Wireless Sensor Networks*, Stockholm, Sweden, June 2005.

[24] A. Dunkels. Protothreads: lightweight, stackless threads in C. [Online]. Available: http://www.sics.se/~adam/pt/

[25] I. Dedinski, H. D. Meer, L. Han, L. Mathy, D. P. Pezaros, J. S. Sventek, and Z. Xiaoying, “Cross-layer peer-to-peer traffic identification and optimization based on active networking,” in *Proc. IWAN 2006 (submitted)*, July 2005.