UNIVERSITY
of
GLASGOW

Application and Network Traffic Correlation of Grid Applications

Jonathan Paisley

September 28, 2006

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy at the University of Glasgow

Department of Computing Science
University of Glasgow

Abstract

Dynamic engineering of application-specific network traffic is becoming more important for applications that consume large amounts of network resources, in particular, bandwidth. Since traditional traffic engineering approaches are static they cannot address this trend; hence there is a need for real-time traffic classification to enable dynamic traffic engineering.

A packet flow monitor has been developed that operates at full Gigabit Ethernet line rate, reassembling all TCP flows in real-time. The monitor can be used to classify and analyse both plain text and encrypted application traffic.

This dissertation shows, under reasonable assumptions, 100% accuracy for the detection of bulk data traffic for applications when control traffic is clear text and also 100% accuracy for encrypted GridFTP file transfers when data channels are authenticated. For non-authenticated GridFTP data channels, 100% accuracy is also achieved, provided the transferred files are tens of megabytes or more in size. The monitor is able to identify bulk flows resulting from clear text control protocols before they begin. Bulk flows resulting from encrypted GridFTP control sessions are identified before the onset of bulk data (with data channel authentication) or within two seconds (without data channel authentication). Finally, the system is able to deliver an event to a local publish/subscribe server within 1 ms of identification within the monitor. Therefore, the event delivery introduces negligible delay in the ability of the network management system to react to the event.

Acknowledgements

I would like to thank the following people for their help and support during the development of this dissertation:

My supervisors Joe Sventek and Peter Dickman for all their advice and patience, and particularly for their support while I have been working away from Glasgow.

The p2popt project for allowing me to make use of their Gigemon hardware, and also the project members at Lancaster University for providing access to remote computing resources.

The systems team at DCS for letting me at their computers, and all the people involved in setting up the Gigemon at the department router, in particular Pete Bailey, Colin Cooper, Chris Edwards, Rolly Gilmour, and Douglas MacFarlane.

My DCS colleagues, particularly Alistair Hutton, for tea and banter, and for listening to my brain dumps.

The yoda of OmniGraffle, Peter McMaster, for his extensive diagram consultancy, and for patiently advising while I was working at home.

My wonderful family: my parents for satisfying my thirst for wires and computery things as a child, including letting me wire up the sofa; my brothers Mike and Matt for humouring my techy obsession.

And finally, Deborah. For her delectable cooking, surest sense of 'rightness' and objective advice. For being truly wonderful.

Contents

List of Figures

List of Tables

List of Listings

List of Inserts

Chapter 1

Introduction

1.1 Background

The practice of scientific research is increasingly associated with the extraction of information from exponentially increasing volumes of experimental data [69]; 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, and significant financial support is provided by a number of governmental organisations (e.g., UK e-Science [1], EGEE [2], TeraGrid [5]) to facilitate this practice.

Scientific research typically requires analysis and correlation of multiple experimental 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 [10], 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 development of high-speed TCP variants [56, 98] can only exacerbate the situation.

Research activities that use distributed network and computing resources in this manner have become known as 'Grid computing', a term which embodies the idea of dynamically managed, wide-area distributed computing. One definition is:

> *[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.* [27]

Although these 'Grid' systems represent a wide range of application types and corresponding network usage (for example, computationally driven experiments may use very little network resources but significant processing resources), those involving bulk data transfers are the most relevant in this work because they dominate the network load. As more and more Grid-style projects move into production, the effect on the network will only increase.

1.2 Current Practice

Most operators of academic research and/or commercial networks will have provisioned their networks sufficiently to handle the growing bulk transfer traffic, often by explicitly provisioning routers and links to carry this type of traffic. This is certainly the case for the latest iteration of the UK's education and research network, SuperJANET5 [102]. Of primary concern to the operators is the impact of prolonged, high-volume, traffic on the quality of service experienced by other users of their networks. Ideally, traffic may be segregated, both to ensure the efficiency of bulk data transfers but also to prevent disruption of other network users. However, segregation requires detection and classification of the transfers.

If the operators are able to detect the onset of bulk transfer activity, they can reengineer the bulk traffic onto specific resources. Unfortunately, the accurate detection of many Grid-style applications is difficult because they do not make use of fixed, well-known transport-level identifiers (such as TCP port numbers). Therefore, Service-Level Agreements (SLAs) for both Grid users and others cannot be maintained by static traffic engineering practises that fail to adequately cope with such dynamic applications.

Traditional traffic engineering approaches involve time-scales of days or more (often significantly longer) to respond to changing demands. Network utilisation monitoring is carried out at too coarse a level to permit the identification of particular flows, and certainly not in time to manage them properly. Therefore, dynamic traffic engineering is needed.

Operators can use MPLS [93, 13] to define multiple paths (Label Switched Paths, in MPLS terminology) between ingress and egress routers within their network, which support differential routing based on factors other than destination address (which is the fundamental principle of IP routing). The ingress router has the task of assigning an incoming packet to an MPLS Forwarding Equivalence Class (FEC) that dictates the subsequent path through the network.

By dynamically modifying the rules that choose the appropriate FEC based on information available in the packet header, real-time traffic engineering can be realised. However, as indicated above, the identification of Grid-style bulk transfers at the network level is non-trivial. Doing so in real-time, such that the identified flows can be re-engineered according to network operator policy, is even more difficult.

1.3 Dynamic Detection

Bulk data transfer protocols define two types of 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.

This dissertation focuses on the dynamic classification and detection of bulk data Grid-style transfer applications by packet payload inspection, specifically addressing the following issues:

1. Signalling and data carried over the same flow.
2. Signalling and data carried over different flows.
3. Detection of such traffic when carried over flows with dynamically assigned TCP port numbers.
4. Classification of such traffic when the signalling data is encrypted.

It is assumed that machine inspection of packet payloads for reasons of network management is not, ultimately, a privacy concern. This assumption is validated by the common use of intrusion detection systems that depend on payload scanning.

This work has been previously reported on, in less detail, in [81]. The work presented in this dissertation and [81] is entirely my own, but I am grateful to Professor Joe Sventek for his assistance in co-authoring [81].

1.4 Thesis Statement

Given the background outlined above, I assert that:

T1. A real-time network monitoring tool can be built that performs full plain text analysis of Grid bulk data traffic at line rate.
T2. This tool can be extended to perform heuristic analysis of encrypted or inaccessible payloads at line-rate.
T3. Such analysis can generate events early enough to enable dynamic traffic re-engineering.

1.5 Organisation of Dissertation

The main body of the dissertation begins in Chapter 2 with the related work. Chapter 3 covers the methodology used to validate the claims above, and in Chapter 4 some specific Grid applications used in the remainder of the work are studied in detail. Chapter 5 describes in detail the development of a real-time monitoring tool for the analysis of the applications of interest, and then plain text and encrypted heuristic protocol analysis techniques are covered in Chapters 6 and 7, respectively. Chapter 8 concludes and covers areas of potential future work.

Chapter 2

Related Work

This chapter examines related work that is relevant to the topic of this dissertation. It begins by covering the necessary background in networking technologies and network management practise. Next, approaches for making network-level measurements are covered and widely used techniques for analysing the resulting data are examined. Finally, the implementation challenges of building systems to handle such analyses are examined.

2.1 Networking Background

This section summarises the main link- and transport-level networking technologies of relevance to the dissertation, and describes the issues involved in the management of today's high-speed internets.

2.1.1 Technology

A key networking technology used in the core of many networks is Ethernet [3], a link-layer protocol for packet exchange. Common deployments are Fast Ethernet (100 Mbits/s) and Gigabit Ethernet (1 Gbits/s). Both of these may be transported over an electrical twisted pair, while Gigabit Ethernet also supports an optical fibre medium (a pair of fibres are used to achieve a full-duplex link). Fast Ethernet (and also the original 10 Mbits/s Ethernet) are common in local area network (LAN) installations, whereas Gigabit Ethernet is often used as a backbone between switches and routers.

Other carrier-grade link technologies include the range of Optical Carriers (OC): OC-3, OC-12, OC-48, and OC-192, with unidirectional speeds of 155 Mbits/s, 622 Mbits/s, 2.5 Gbits/s, and 9.6 Gbits/s, respectively.

The Internet Protocol (IP) can be carried by any of the above link-layer protocols. In turn, the IP protocol can carry Transmission Control Protocol (TCP) [88] packets. TCP is a reliable transport mechanism that provides a bidirectional stream interface for communication between two end points. It adaptively adjusts the

transmission rate in response to network losses, and multiple TCP channels operating on a shared link will share the available bandwidth proportionally. The TCP 'slow start' mechanism allows a connection to slowly probe the network to determine the available capacity, avoiding congestion from an inappropriately large burst of data. Although this can be a problem for short-lived connections which fail to reach the full bandwidth usage before they are terminated, it tends to be insignificant for long-lived bulk data transfers.

In today's very high bandwidth Internet paths—as used by large scientific research projects—standard TCP suffers for several reasons. It uses a window mechanism to limit the amount of data in-transit on the network that has not been acknowledged by the recipient. For most efficient use of the network, the links involved in the route between the end hosts should be constantly fully utilised. This requires a window size equal to the product of the link bandwidth and the transmission delay, known as the bandwidth-delay-product (BDP). A path with very high bandwidth, very long delays, or both, may have a BDP that exceeds the maximum allowable window size in TCP, which is 64 Kbytes. Extensions for high-performance TCP have been established [55] to alleviate this problem, by introducing *scaled windows*, which increase the window size to 1 Gbyte.

Unfortunately, with such large window sizes standard TCP performs poorly in the event of packet losses because they have a drastic effect on throughput. A pragmatic approach that can help—and also 'steal' a larger share of bandwidth—is to make bulk data transfers using parallel TCP streams. The detrimental results of losses are then distributed among the streams, thus improving throughput.

Several alternative implementations of the TCP congestion control algorithm have been proposed, such as TCP Vegas [23], FAST TCP [56], H-TCP [98], HighSpeed TCP [45] and Scalable TCP [61]. Vegas and FAST both use a delay-based approach rather than loss as a measure of congestion. H-TCP and Scalable TCP, which builds on HighSpeed TCP, make alterations to the congestion window update algorithm to better support high BDP links. All of these TCP variations and are intended to co-exist with other traffic using traditional TCP.

Beyond modifications to the TCP protocol to ensure high performance across large BDP links, improvements to end-host TCP and application implementation can be necessary to avoid processing bottlenecks. For example, use of zero-copy sockets (where packets are received by the network card and made directly available to a user process without the overhead of copying) and checksum offloading (the network card deals with checksum calculation) have been proposed [31].

2.1.2 Management

Management of a network involves both planning for future demand and ensuring that the day-to-day operation of the network falls within the levels of service agreed upon between an ISP and its customers. Failure to meet these levels could lead to

significant penalties, depending on the nature of the agreements made.

The process of managing the flow of traffic through a network is known as *traffic engineering* [12]. This takes place over a range of timescales. For example: capacity planning would be undertaken over periods ranging from days to years; manual routing parameter adjustments from minutes to days; and packet-level processing from picoseconds to microseconds. The engineering process depends on having sufficient capacity and the ability to effectively control routes, queues and anything else that can affect the traffic flow. The process used to make control decisions may involve forecasting future demand or monitoring in real time to identify actual link utilisation. Here, 'real-time' can simply mean humans monitoring the network for congestion and attempting resolution on an ad-hoc basis.

Creating plans to cope with expected traffic demands is time consuming, and large changes have the potential to cause disruption. Therefore, a significant challenge is to achieve automated control processes that can adapt quickly without being too costly or causing instability.

Diffserv[1] [19] is a standard technique for specifying Quality of Service (QoS) parameters for a flow, where an edge router may classify packets by setting or reusing the Type of Service (TOS) field in the IP header. Internal routers can then apply differential routing depending on this field in addition to the destination IP address, according to some predefined QoS policy.

As previously described in Chapter 1, Multi Protocol Label Switching (MPLS) [93], is an alternative technique that avoids the need for destination-based routing within the network core. Packets are still classified at an edge router (a Label Switching Router), but instead of setting TOS bits, a Forwarding Equivalence Class is assigned that dictates the path through the network. Such paths are known as Label Switched Paths (LSPs) and can be configured with different QoS guarantees, such as bandwidth, delay or loss. An advantage of MPLS over traditional IP routing is that the forwarding decision at each node can be much simpler than the potentially complex longest-prefix matching required by the IP protocol. MPLS simply requires indexing into a table using a 20-bit label field. By adjusting LSPs and classification rules, MPLS may be used for effective traffic engineering [13].

The long-term network planning activities described earlier generally manage to deal with long-term trends in traffic, but specific load changes or burstiness (perhaps over several minutes or hours) are not accommodated. Given some mechanism for identifying the current state of the network, traffic can be re-engineered by updating routing tables, adjusting LSPs, or changing the rules for Diffserv or FEC mapping. Rondo [7] is an automated system intended to manage congestion in core networks in near real time (around 30 seconds to one minute). Based on the

1. Contrast with Intserv, which requires applications to specify their requirements [22] and router support along the entire path.

results from a series of network probes, it reroutes Label Switched Paths to reduce the number of overloaded links. It does not take into account the specific causes of congestion (for example, a particular set of bulk data flows), dealing instead with the aggregated LSPs that are responsible for congestion.

2.1.3 Summary

Gigabit Ethernet is an ideal technology for the development of network monitoring techniques both because appropriate hardware is readily available, and because it is frequently used at the edge of an ISP's network, where the kind of monitoring system dealt with in this dissertation would be deployed. MPLS traffic re-engineering represents an effective way to dynamically manage specific flows due to its independence from the IP routing protocol.

2.2 Measurement

Measurements are used to identify actual performance levels in a network. An overall goal is to increase the performance of the network for its users. Measurement may be undertaken within an ISP's network to effectively inform traffic engineering, as discussed in the previous section, or by end users, administrators, and researchers to understand and resolve network performance issues (perhaps by adjusting configuration parameters or protocols). Measurements are also important for straightforward tasks such as billing and customer reports.

Network measurements can be divided into two distinct categories: active and passive. Active measurements inject synthetic traffic into the network to observe network performance. Some infrastructure is required at both endpoints[2]. Passive measurements observe the actual traffic on a network link. Two-point passive measurements can be used to measure one-day characteristics; one example, that of delay, requires that both measurement points have precisely synchronised clocks (e.g., using Global Position System receivers).

2.2.1 Active

Active measurements can be undertaken relatively easily using straightforward tools. They may involve the determination of round-trip times, available bandwidth, IP routing, and loss. There are many deployed systems designed to gather such measurements over long time periods. For example, the NIMI (National Internet Measurement Infrastructure) [86] uses a distributed set of monitoring systems supporting generic end-to-end, on-demand, monitoring. Other specific projects include the RIPE NCC[3] Test Traffic Measurement project [91, 46], NLANR's (National

2. Although this may just mean an Internet host that is willing to respond to ping messages.
3. Réseaux IP Europeéns Network Coordination Center

Laboratory for Applied Network Research) NAI (Network Analysis Infrastructure) [73] and PingER [70].

Other large-scale measurement projects, such as Surveyor [57] and AMP [74] use similar techniques involving GPS-synchronised clocks for one-way measurements. In contrast, the PingER[4] project from the IEPM group at the Stanford Linear Accelerator Center (SLAC) [70, 54] makes use of the `ping` tool for performing RTT measurements. All of these projects involve large numbers of geographically distributed hosts (typically > 50) which form a testing matrix. In the case of the PingER project, for example, ten 100-byte pings are sent at one second intervals, followed by ten 1000-byte pings at one second intervals. These measurements are repeated every 30 minutes.

2.2.2 Passive

Although active measurements can report on properties of the Internet paths between end hosts, they cannot yield information about the performance of specific flows, nor aggregated properties of the traffic, such as the mix of flow durations or the distribution of traffic according to the application responsible for it. These types of measurements require inspection of the actual packets present on a link. Compared to active measurements, the costs can be greater due to the need for continuous collection of data on a link operating at full load.

Existing network infrastructure may be used to perform passive monitoring, by leveraging such information as interface packet counters maintained by a switch or router. The Simple Network Management Protocol (SNMP) [28] is a widely used and supported protocol for retrieving this kind of information and making configuration changes. However, the interface is fairly low-level and suffers from high overheads that make it inefficient for the transport of large amounts of monitoring data.

The Internet Engineering Task Force (IETF) define a series of Management Information Bases (MIBs) that specify a schema for the extraction of information from an SNMP-enabled network device. Vendors implement the MIB specification on their hardware, which in turn enables monitoring systems to retrieve measurements using a standard interface. RMON [103] is one such SNMP definition for remote network monitoring devices. In addition to basic interface statistics (such as byte counts and packet counts), RMON specifies counters that are aggregated by hosts, top-N-hosts, host sets and also arbitrary filters. Although hardware and software vendors have implemented RMON-enabled systems, usually only a subset of the specification is supported.

The IETF has also produced an architecture document [26] defining methods to specify flows and standard metrics for flow measurement, with the intention

4. End-to-end Reporting.

that output from multiple vendors' equipment/software will be interoperable. A corresponding SNMP MIB is also given [25].

NetFlow [32] is a mechanism implemented by Cisco routers that tracks traffic on a per-flow basis. This means that it can provide finer-grained data than plain SNMP, but does not provide detail down to the individual packet level. A router periodically flushes NetFlow records to a collection host via UDP (which runs, for example `cflowd` [30]). Performance limitations usually dictate that NetFlow-style records are based on sampled flows. Most packets are processed by a router's fast path, which precludes the updating of counters. These sampled flow statistics are not ideal for billing purposes, nor do they give an accurate view of network usage.

2.2.3 Summary

Whilst active measurements can provide a great deal of useful information on the performance characteristics of a network, they are not effective in identifying specific traffic features, such as those caused by bulk data transfers. Aggregate data collection techniques like NetFlow can identify specific flow-level features, but their granularity means that they are insufficient to allow the identification of specific applications and subsequent dynamic re-engineering during a flow's lifetime. Therefore, finer-grained passive measurements are required.

2.3 Analysis

This section considers the techniques used in the analysis of passively gathered network traffic information (which includes both full packet traces and aggregated statistics). Network traffic 'analysis' covers a broad range of activities, for example: simple accounting based solely on packet headers; intrusion detection; network debugging; and traffic classification (i.e., identifying the application responsible for a particular flow). Alongside these different reasons for carrying out analysis there are also different approaches. For example:

- Inspection of packet headers (e.g., IP addresses and well known ports).
- String matching within individual packet payloads.
- TCP stream reassembly for application-level content analysis.
- Time series analysis of packet arrival times [53].
- Statistical inferences based on flow properties.

Some of these techniques are suited to real-time operation (e.g., header inspection or string matching) because of the bounded amount of work that needs to be done per-packet. Others, due to their statistical nature, require longer time periods to yield good results and are typically carried out on long packet traces. TCP stream

reassembly is complicated by the necessity for application-specific support to enable protocol interpretation, and for the requirement of effective buffer management to construct an ordered set of packets/bytes.

Aggregate flow monitoring tools such as CoralReef [62] (see Section 2.3.4) use simple packet header inspection to categorise traffic according to IP addresses, network prefixes and port numbers. Summary statistics from this kind of analysis can be useful for reporting on broad-scale network usage patterns. For example, the Internet2 Netflow Weekly Reports [4] present a range of tables and graphs showing packet and byte counts over a week-long period, split between bulk and non-bulk traffic (where 'bulk' is defined to be a flow that transfers more than 10 Mbytes of data). Measurement traffic such as iperf (see Section 4.7 for more details on this application) frequently accounts for around 40% of the bulk data by volume. The reports also show traffic categorised by application, which has been determined by a fixed port-number based classification scheme. Although applications such as GridFTP and BBFTP (see Chapter 4 for more details) are represented, they appear to account for a very small proportion of the traffic. This is because the port-number technique is only able to identify the control connection for a file transfer, and then only if the software is configured with default port numbers. Therefore, it is likely that a significant proportion of traffic corresponding to these applications is being under-estimated.

2.3.1 Classification

The term 'classification' can take two different meanings in the context of traffic analysis. Either the traffic of interest naturally belongs to some class (e.g., a data connection corresponding to one of the applications mentioned above) and the problem is to correctly identify this from the available network properties, or else a categorisation is imposed by the investigator because it is useful for traffic engineering or other purposes, such as distinguishing between 'elephants and mice' [83]. The selection of a categorisation technique for identifying traffic ultimately depends on the context in which it is used. For example, the Internet2 weekly reports [4] use a 10 Mbytes volume threshold for 'bulk' identification. In other situations it might, for example, be preferable to employ a throughput-dependent threshold, perhaps expressed as a fraction of the total link bandwidth.

The application signature approach to application identification [96, 67] 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 behaviour 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 usually results in overload for the soft-

ware, resulting in dropped packets. Finally, some application protocols avoid payload inspection by using encryption algorithms (see Section 2.3.3).

Transport layer port identification [97] 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 the 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 [58].

Heuristic-based network/transport layer approaches [58, 59] 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. These methods can give good performance for existing application protocols and may even be used to discover unknown protocols. However, the validation of such methods is a challenge.

Moore and Papagiannaki [77] have developed a framework for identifying the application corresponding to network flows by a series of nine increasingly complex classification techniques. Each technique builds upon the results of the less complex. These techniques range from straight-forward port-based classification through single packet signature recognition, signature recognition on the first KByte of a flow, protocol recognition on the first KByte and analysis of the entire control flow. Entire-flow analysis (for example, with FTP control channels) allows the identification of associated flows (the data flows).

The results of this offline classification process have been used to aid in the validation of heuristic techniques for application identification. For example, [78] uses supervised machine learning to illustrate the application of Bayesian analysis techniques to traffic classification, which categorises traffic into one of several equivalence classes[5]. The pre-classified data set enabled the evaluation of the technique, which identified several important traffic discriminator properties, such as the number of pushed data packets and the average segment size across a flow. Similar results and approaches are reported in [107, 108, 72, 17]. Work is ongoing in identifying the effectiveness of these techniques and, in particular, their applicability to real-time classification.

BLINC [60] is a different approach to classifying traffic flows that is based on observing patterns of host (rather than individual flow) behaviour at the transport level. A host's behaviour is examined in terms of its interaction with other hosts, capturing communities of nodes and whether a host acts as a provider or consumer of a service, or both. A similar, but less general, approach has been used in combination with payload signatures in the identification of peer-to-peer traffic

5. Such as bulk, interactive, mail, www, peer-to-peer, attack, and multimedia.

[59].

2.3.2 Application Level

Application-level traffic analysis refers to the practise of interpreting the contents of a flow in terms of the application-level protocol structure. This requires that a flow has already been correctly classified.

In the analysis of the Skype peer-to-peer telephony protocol [15], detailed investigations using tools such as `ethereal` (see Section 2.3.4) were undertaken, yielding insights into the operation of the proprietary protocol. The results of this kind of work can be used to inform the development of an automated analysis system.

As mentioned in the previous section, application-level analysis can also be used to inform the classification of *other* flows that relate to the flow under study. A typical example of this is the identification of the FTP data channels set up by an FTP control channel.

The intrusion detection system Bro (see Section 2.4.3) also uses application-level protocol reconstruction.

2.3.3 Encryption

Encrypted flows preclude content inspection for traffic analysis. Instead, some approaches based on timing or packet size may yield useful information.

[99] presents a technique for reducing the scope of the search required for a brute-force password attack (by about one bit per character pair for randomly chosen passwords). It uses the inter-packet timings resulting from each key of a password being entered over a secure shell (SSH) connection as well as the size information leaked due to padding by the encryption algorithm.

A similar, size-based, attack against HTTPS (secure web connections) is presented in [51]. HTTPS leaks the approximate size of objects transferred (e.g., web pages and referenced images), which means that a correlation between the actual sizes of the resources present on a particular website and the leaked sizes can be used as evidence that a particular site is being accessed. [18] extends the analysis to exploit the statistical characteristics of HTTP connections that are encrypted as a single flow (e.g., using WEP/WPA[6], IPsec and SSH tunnels).

2.3.4 Tools

Several tools are available to support the manual inspection of packet contents in order to enable protocol understanding. The familiar `tcpdump` tool provides a textual representations of packets, along with decoders for some common protocols. The open-source `ethereal` [34] project offers a graphical user interface along with

6. These are wireless encryption standards.

a comprehensive range of protocol decoders. These decoders present protocol features in a simple display that can be used to browse a trace file, and each decoded element can be used in a filter expression to limit the set of packets being viewed. A TCP reassembly feature means that a particular stream can be quickly identified and its contents viewed. NetDude [63] presents a similar graphical interface to `ethereal`, but focuses on the efficient viewing and editing of large trace files.

Tools such as these are vital in the development of network analysis techniques, since they can be used to improve the understanding of a protocol and examine packet-level properties responsible for particular features, which might be encountered during an investigation.

The MAGNeT [43] system is a monitor for application-level behaviour, which operates by hooking into the Linux kernel to trace system calls and the flow of data through the network stack. The resulting data could be used to drive different implementations of network protocols to test new designs, or analyse unexpected interactions between application and protocol behaviour that are the cause of poor performance. Similar information can be obtained from passively monitoring at the network level, but inferring the original application-level operations is much more difficult.

The multi-protocol visualisation tool demonstrated in [50] builds upon the Nprobe project [76] (see Section 2.4.2) and `tcptrace` tool [80] to visualise the correlation between TCP-level and HTTP-level behaviour during a web page download (along with associated resources such as images). Such visualisations can be used to inform the development of an automatic mechanism for identification of specific application/network features.

At a higher level than these systems for inspection of packet properties are tools such as NeTraMet [24] and CoralReef [62], which can be used to automate the extraction of aggregate properties of network traces and real-time links. CoralReef is a passive traffic monitoring suite consisting of drivers and applications for packet capture and real-time report generation. A variety of data sources can be used, including NetFlow records, `libpcap` and native capture using a DAG card (see Section 2.4.1). The tools focus on flow-level statistics, identifying traffic by host addresses and port numbers and providing summary reporting of variables such as packet counts, byte counts and other flow characteristics. For passive analysis research, these kinds of measures can be very useful in identifying significant features of a trace for further investigation.

2.3.5 Summary

Monitoring for application-level traffic classification and timely event generation at Gigabit Ethernet line rates requires use of appropriate techniques in order to achieve real-time performance. Signature scanning is effective, but becomes costly when applied to a large proportion of packets. Therefore, it is important to iden-

tify header-level properties (such as packet size or specific TCP flags) to reduce the candidate set. Heuristic analysis techniques that depend on long time scales or whole-flow properties are inappropriate for small time-scale classification purposes. Application-level approaches enable the extraction of additional information that may be used to identify related flows (e.g., FTP control and data), but require careful implementation to operate efficiently.

2.4 Monitoring

This section examines the range of approaches available for performing passive network monitoring. Most of the techniques covered here involve attachment to an operational network via some sort of passive tap mechanism. Low-speed Ethernets using a central hub can simply be monitored by attaching to the hub. Some managed switches have the facility for 'port mirroring' where all frames are delivered to a special mirror port. However, this can lead to performance degradation of the switch and skewed timing, and a single transmit channel on an Ethernet interface is insufficient to carry full-duplex traffic (nor the potentially greater bandwidth of the switch's backplane). The preferred solution is to use an optical splitter, which transmits packets both to the original destination and a monitoring interface. Provided it is installed correctly and there is sufficient optical energy to maintain error-free reception, the optical splitter can operate completely passively. The main drawback is the downtime required to install the splitter.

2.4.1 Packet Level

In 1996 the OC3Mon [11] was a pioneer in high-speed passive network monitoring, using two ATM network cards in a PC on an optically tapped link. The goal was to avoid complicated statistical calculations by ensuring that every packet could be captured (i.e., no sampling was required). Packets could either be recorded into memory and later dumped to disk, or real-time flow-level aggregation could be performed. Customised firmware on the ATM cards handled a double-buffering scheme, where the host software would process a 1 Mbyte buffer while the card filled (via PCI bus-mastering) another 1 Mbyte buffer. An interrupt from the card once a buffer was filled indicated that the software should begin processing. In order to achieve the greatest performance, the system was built upon DOS instead of a Unix operating system. This decision was determined by the ease with which the application could monopolise the CPU and memory, thus eliminating any complications arising from scheduling anomalies or memory fragmentation (the network interface cards required access to physically contiguous memory regions).

Several years later the OC3Mon project has evolved into a commercial enterprise by Endace Measurement Systems, who produce a range of custom-built

hardware network monitoring cards. The 'DAG' hardware series [33] is capable of full-duplex, full-payload, line-rate monitoring at up to OC-48 speeds. Accurate time-stamping of captured packets is handled by the hardware at packet reception time, and the onboard clock can be accurately maintained via connection to an external GPS signal. The double-buffering scheme employed by OC3Mon has been extended to a full circular buffer, and the hardware is compatible with both Linux and Windows operating systems. Recent developments have produced a co-processor add-on [40] which is capable of filtering packets by header inspection before they are delivered to the application.

While the DAG card represents a logical evolution of the OC3Mon project, in terms of increasing the capabilities of the hardware to alleviate software-related issues, several projects have tackled the problem of monitoring modern high-speed networks using careful software techniques and commodity hardware. Before examining these in detail, it is necessary to highlight the issues surrounding efficient packet capture and processing.

The classic network Unix network monitoring tool `tcpdump` [64], which is based on the `libpcap` [65] library, is easy to use but design limitations mean that it is unsuitable for high-speed capture.

`tcpdump` and `libpcap`'s architecture involve multiple copies of packet data between network card, kernel space, and user space. On some platforms (`libpcap` is supported on multiple Unix variants) a system call is required for each received packet, which adds significant overhead, especially when large numbers of small packets are involved. In addition to a system call per-packet, the network interface card may also generate an interrupt for each packet. This can lead to the kernel spending considerable time servicing interrupts. Packet time-stamps are generated by calling `gettimeofday` after each packet is received, which means that process scheduling latencies can compound the time-stamp error (which is of the order of 1 ms). Simple variations on `libpcap` using a user/kernel space shared ring buffer [38, 106] can reduce the system call overhead but still require at least one copy of each packet.

To overcome the problems of achieving line-rate capture and accurate time-stamping[7], one or more of the following techniques have been employed: a standard or modified open-source Unix operating system (Linux/FreeBSD/NetBSD); network card supporting on-card time-stamping; multiplexing of received traffic across several monitoring hosts; and GPS receivers for high-accuracy timing and NTP for coarse synchronisation.

TICKET [105] uses two or more commodity PCs, one or more for packet capture and one for processing. The passive monitoring capture machine(s) use a stripped down Linux kernel whose `init` process has been modified to run only

7. Pásztor and Veitch [84] provide a detailed analysis of these timing issues.

the monitoring software and without leaving kernel mode (this is a similar to OC3Mon's approach of using DOS to drive the monitor). Time-stamps are made in the kernel as soon as the packet is received from the network card, although it is not clear how accurate these are. Packet headers only are buffered and passed to the secondary processing host over a dedicated network interface. The load-sharing features of a Gigabit Ethernet switch were used to distribute the load across two capture machines, which enabled packet header monitoring at Gigabit speed.

Linuxflow [66] also uses a modified Linux kernel to achieve its goal, which is a cost-effective network management tool capable of high-speed traffic accounting. A new Unix socket type is introduced to allow packet reception to bypass the normal network stack for delivery directly to a user-space application. The multi-threaded application is responsible for reading packet data from the kernel, aggregating packets into flow records, and sending the aggregated data over the network to a collection system. The authors report that a single four-processor, 64-bit host is able to capture without loss at up to around 1 Gbits/s (i.e., half-duplex Gigabit links).

`pktd` [47] is a packet capture/injection daemon, intended to be the sole trusted, privileged entity needed by a host that provides external measurement services. It multiplexes access to the network device, with different rights per user as assigned by the system administrator. The Self-Configuring Network Monitor (SCNM) project [6] uses `pktd` to implement a passive monitoring infrastructure for packet headers that can be activated by end-users. Special UDP activation packets cause the `pktd` filter to be updated to include the relevant host pair, and subsequently a corresponding trace is sent to one of the end hosts. A typical deployment would involve SCNM measurement systems at two or more points along the network route. This supports the identification of which segments of the network are the source of problems for an application data stream. The authors report that the hardware is capable of capturing all packets on an 80% utilised Gigabit Ethernet with an average packet size of 800 bytes.

Although it was based on `pktd`, the SCNM project made some alterations to the FreeBSD kernel to extract time-stamp information from a hardware register in the Gigabit Ethernet network card (usually the system clock time when the kernel processes the packet is used). `pktd` is based on `libpcap`, which provides a packet filter mechanism based on the Berkeley Packet Filter (BPF) [71]. BPF supports arbitrary header-based filtering using a bytecode program that is executed in the kernel. A simple filter language specifying such criteria as host addresses, port numbers, packet size or TCP flags is compiled by `libpcap` and passed to the kernel. The kernel checks that the bytecode is safe (that it has no backward branches that might cause an unbounded loop) before accepting it. BPF suffers from the necessity to filter packets more than once. For example, once the in-kernel filter has been evaluated, an application process may have to make similar header inspec-

tions in order to carry out its task.

The goals of the SCAMPI [35] project in 2003 were to produce a high-speed network monitoring platform that specified a standard, efficient Monitoring API (MAPI) to replace `libpcap`, to improve upon the expressive power provided by current filtering mechanisms such as BPF, and to support scalability through optional use of special-purpose hardware. An example of the kind of additional expressive power desired is the ability to specify sampling, perform TCP reassembly, or automatically extract aggregates (packet/byte counters etc) from flows. A later implementation [87] using both a standard Gigabit Ethernet network card and DAG card (described earlier) was presented. The authors give an example of the identification of ephemeral FTP data ports using MAPI and substring searches within packets. However, they indicate that the maximum number of loss-free flows that can be processed is only 100 at 700 Mbits/s. MAPI emphasises the flexibility of multiple applications accessing the packet stream at once, and it appears that in this implementation the flexibility comes at a cost.

A different implementation of the SCAMPI architecture was FFPF (Fairly Fast Packet Filters) [20], which replaces BPF (and its Linux variant, the Linux Socket Filter), providing compatible implementations of both MAPI and `libpcap`.

FFPF can automatically allocate filters to the best place in the processing hierarchy (network interface, kernel and user space). An implementation on the IXP1200 network processor supports filtering and buffering on-board. A consideration in any system using memory on an expansion card (connected via the PCI bus, for example) is whether the packet data should be copied. Three options are available: leave packet data on the card and access it over the PCI bus; copy selected packets to the host memory via bus-mastering; or copy all packets. For example, if the host needs to examine the packet extensively then in the first case most reads will have to cross the PCI bus, which is expensive. FFPF lets the administrator decide at runtime which approach to use. While the filtering mechanisms provided by FFPF allow independent applications to process the monitored packets, the performance of the implementation (in kernel and on the IXP1200) and dependence on essentially static filters to select relevant traffic makes it unsuitable for dynamic traffic classification.

2.4.2 Flow Level

Several projects have focused on the correlation between network- and application-level traffic. BLT [42] used examination of HTTP/TCP traces taken using PacketScope [9]. Traces were stored to disk and periodically processed according to a simple pipe-lined process. Rather than precisely reconstructing TCP flows (which was difficult in the face of some packet loss by the monitoring device), a broader technique was developed using the steps of IP flow demultiplexing, TCP sequence number sorting, duplicate elimination and loss identification. From the resulting

data stream, HTTP protocol information was extracted by scanning for well-known signatures (such as `GET` requests). This processing was done in real-time, due to limitations of disk and tape storage.

An alternative and more general approach taken by the Nprobe [76] project involves the real-time extraction of protocol-specific features, for online analysis of protocol interactions and reducing the quantity of data that has to be stored on disk for a trace. Protocol-specific knowledge is used to save only the information relevant for later study. Nprobe uses several of the techniques encountered in the previous section to achieve high performance and accuracy, including time-stamps and filtering by the network interface card, interrupt coalescing, kernel modifications for efficient packet access by user-level processes, and striping across monitoring hosts[8]. Performance results indicate that the system, when operating on a synthetic HTTP load, is able to cope with 165,000 packets per second (for small HTTP transactions) and 44,000 concurrent flows at 304 Mbits/s (117,000 packets per second). These limits are due to a combination of memory usage and CPU time taken to manipulate per-flow state.

The Windmill project [68, 104] had similar high-level goals to a combination of SCAMPI and BLT, in that a broad range of measurement techniques should be supported for multiple concurrent applications, including application-level protocol reconstruction and correlation with the underlying transport protocol's events. Windmill specifies a packet filter architecture, WPF, that addresses the performance problems of filtering being performed both at the kernel and application level. The output of a WPF filter includes the identity of the application/module that should process the packet, so that subsequent filtering is not required. Protocol modules were implemented to handle TCP reconstruction and analysis of application-level conversations. Since for some application modules it may not be necessary to reconstruct a full TCP stream (perhaps only examining header and length fields), the TCP module could be reconfigured to avoid the expensive TCP reassembly process.

The above projects implement TCP reassembly in the monitoring host to enable application-level traffic investigation. BLT used a somewhat ad-hoc approach that was appropriate for the particular task of examining HTTP traffic, especially given the hardware limitations at the time. Nprobe, Windmill, SCAMPI and FFPF all are capable of TCP reassembly, but the details of the particularly implementations are not given. One significant problem is the potential for a large amount of buffering required in the monitoring system for flows with large window sizes (see Section 2.1.1). A novel hardware-based system, TCP-Splitter [94], employs an active/passive hybrid technique that is deployed in-line with a network link. The hardware has both an input and output interface operating at OC-48 rates. Packets are passed from input to output via an FPGA (Field Programmable Gate Array) that

8. An n-valued hash of packet header fields is used to decide which monitor machine will accept a packet. A drawback of this approach is its inability to cope with a single high-bandwidth flow.

selectively drops out-of-order packets from TCP connections[9]. The consequence of such actions is that the retransmission mechanism in the end hosts will send the next in-sequence segment. Therefore, an in-order stream of packets can be passed to the host computer for further monitoring, with the crucial property that buffering for TCP reassembly will not be required. The authors claim that actively dropping frames may actually improve TCP performance for end hosts using a Go-Back-N retransmission policy.

2.4.3 Intrusion Detection

Another class of network monitoring devices, which have a specific purpose rather than the general aims of most of the projects previously mentioned, are Network Intrusion Detection Systems (NIDS). They scan for malicious traffic patterns in order to report to an operator the incidence of network-based attacks. Such a system would usually be deployed at a gateway link within an organisation, rather than at an ISP. Section 2.3 detailed the common techniques used. This section examines the properties of two specific NIDS: Snort [92] and Bro [85]. Snort is a signature-based detection system that employs a library of patterns (which is continually kept up-to-date) against which packets are matched whereas Bro focuses on application-level protocol interpretation. In Snort, the patterns are first matched against network-masked host addresses and port numbers, and subsequently a rule can restrict itself to specific IP/UDP/TCP header field values or perform content inspection by string scanning. Since string scanning is an expensive operation, it is important that any additional criteria limit the number of packets to search.

Snort also offers a TCP reassembly module that enables pattern matching across packet boundaries. The module documentation indicates that it can operate over 'several thousand simultaneous connections'. While this level of performance may be adequate for a small organisation, it is certainly insufficient to cope with the level of traffic that might be expected at an ISP's edge router. An active hardware-assisted system using similar techniques to Snort is CardGuard [21]. Designed to operate at Fast Ethernet (100 Mbits/s) speeds for a single host or small group of hosts, it uses an IXP1200 network processor to perform online intrusion detection to block malicious traffic. TCP flows are reconstructed on the network processor, with the limitation that any buffering overflow causes the flow to be conservatively dropped, i.e., non-malicious traffic may be completely blocked.

A substantially different approach is taken by Bro. It is designed to operate at Gigabit Ethernet line rate, and depends on `libpcap` for access to network traffic. Bro employs full-protocol analysis, which is accomplished by fully reassembling the monitored TCP streams (taking care to achieve the same reassembly as an

9. This requires a fixed-size TCP connection hash table to be maintained in hardware. Collisions have the undesirable property that packets from different flows will be treated as if they were the same connection.

end host, in the face of malicious traffic), classifying them by application according to port numbers, and passing the incoming data to the appropriate application-specific protocol interpreter module.

Once a protocol interpreter has identified an event of interest (for example, a Finger protocol query for a particular user name), an event is generated that is then handled by a *policy script*. The policy script is written in a Bro-specific interpreted language and can be modified by the end user. This supports the separation of the identification mechanism (in native C++) from the action policy. Indeed, Bro has been extended for application to the task of network trace anonymisation by using special policy scripts that mask sensitive application-level elements, whilst preserving as much of the conversation as possible [82].

To avoid heavy processing (e.g., stream reassembly) overhead for unknown TCP streams, Bro employs a packet filter to limit the packets under study to those involving well-known port numbers for which a protocol analyser is installed. The packet filter is also augmented to pass all TCP SYN and FIN packets, therefore enabling the monitoring of protocol independent security incidents such as SYN flooding or port scanning.

Like Snort, Bro operates most effectively near the edge of the Internet where it can be used to monitor traffic at the organisation level. In order to achieve reasonable performance it uses statically determined well-known-port filters, which limit its ability to monitor potentially malicious traffic operating on unusual ports.

2.4.4 Summary

Flow-level traffic analysis techniques involving payload inspection allow the specific identification of control flows and their corresponding bulk data flows, which means that bulk data encountered at the network level can be attributed to a particular application (and therefore can be covered by a traffic management policy). Approaches that offer single-flow classification without reference to payload content cannot accurately identify these associated bulk flows.

While there has been some success in producing line-rate monitoring systems using commodity equipment, easy-to-use dedicated network monitoring hardware is now available. Such technology means that an investigator can focus on network analysis tasks with loss-free packet capture, rather than be concerned with the complexities of attaining such performance.

2.5 Conclusions

This chapter has examined work relevant to the task of classifying bulk data traffic according to the application responsible. Performing such analysis is difficult without the use of control-traffic payload inspection to accurately discover IP addresses

and port numbers used by the bulk data transfers. Although statistical heuristic analysis techniques can be applied to differentiate between broad traffic classes (e.g., elephants and mice) and even more fine-grained equivalence classes (such as bulk, mail, peer-to-peer, and www), they suffer from two main problems. Firstly, they cannot be applied effectively in real time. Secondly, they cannot yet distinguish between bulk data resulting from different applications. Given some other technique for control-traffic classification, hand-crafted heuristic techniques can be effective for dealing with encrypted traffic.

Advances in network monitoring hardware mean that payload inspection and application-level protocol reconstruction/interpretation are attainable at Gigabit Ethernet line rates. Combining a system of this type with modern traffic engineering technologies, such as MPLS, can enable dynamic traffic re-engineering, and therefore contribute to improved network performance.

The next chapter introduces the methodology used to prove these claims.

Chapter 3

Methodology

This chapter describes the methodology used to prove/disprove the Thesis Statement initially introduced in Section 1.4, namely that:

T1. A real-time network monitoring tool can be built that performs full plain text analysis of Grid bulk data traffic at line rate.
T2. This tool can be extended to perform heuristic analysis of encrypted or inaccessible payloads at line-rate.
T3. Such analysis can generate events early enough to enable dynamic traffic re-engineering.

3.1 Network Infrastructure

In order to carry out the investigation of Grid-style applications, two types of network setup were required to support the necessary experiments: firstly, a small test-bed network for the isolated investigation of application properties; and, secondly, a wide-area network for the testing and evaluation of the completed system.

Within each of these setups, it was necessary to be able to attach the passive network monitoring device at an appropriate location. Figure 3.1 shows the structure of the network elements used.

Two academic institutions are represented: Glasgow University and Lancaster University. These obtain IP connectivity over the UK academic network, JANET. At Glasgow, the main connection point is at the Computing Service, where the link speed is 10 Gbits/s. Within the Glasgow campus, the diagram shows the Bioinformatics and Computing Science (DCS) departments. Note that they achieve connectivity from a Gigabit Ethernet link. A set of six machines in DCS (`nsmc01` to `nsmc06`) were used as the local test-bed. A combination of `nsmcXX` machines and the remote `kosciusko` and `bananarama` (in the Bioinformatics department and Lancaster University, respectively) were used for testing and evaluation.

Shown on the diagram are four points labelled A, B, C, and D. These represent potential locations for the attachment of the passive monitor device. Points A

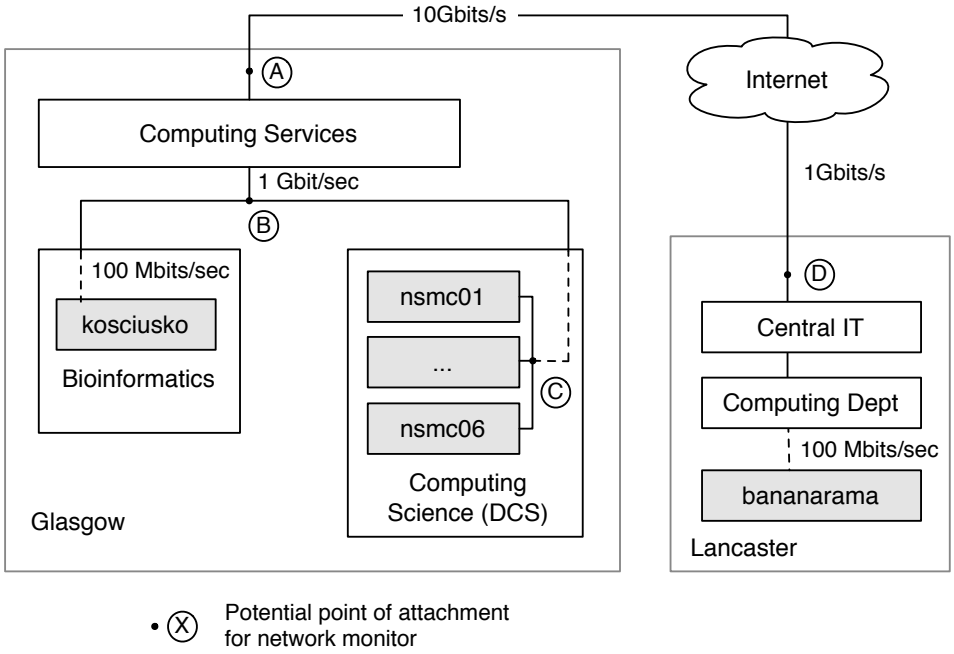


Figure 3.1: Experimental network topology.

and D would observe all traffic to/from the Glasgow or Lancaster campus, respectively. Point B allows all traffic between the Glasgow departments to be monitored, as well as traffic passing from them to the internet (and vice-versa). Finally, point C allows traffic between the test-bed machines to be observed.

During the course of the work, due to local policy and security restrictions as well as the administrative overheads of managing the system at a remote site, the monitoring hardware was only connected at points B and C.

3.2 Line-Rate Data Capture

As discussed in Section 2.4.1, full traces of high-bandwidth network traffic can be captured using the commercially available DAG network monitoring hardware. This solution was chosen because of its efficiency, and particularly because it eliminates any concerns about the reliability that might be encountered from an approach using off-the-shelf networking hardware (such as a standard Gigabit Ethernet network interface card).

The manufacturer of the DAG card (Endace Measurement Systems) also produce a fully integrated rack-mountable PC system known as a GIGEMON, which was used in this work. The GIGEMON incorporates the 64-bit PCI-X DAG card into a dual processor 2.8 GHz Xeon system. The hardware and software can be tuned for capturing full-payload traces to disk or performing real-time content-analysis. This will be demonstrated in Chapter 5.

A tape library was used to archive full payload traces in case the need to revisit an experiment ever arises.

In order to understand the correlation between behaviour of an application both at the application and network level, it was important to be able to gather application-level logs at the same time as the network traces being gathered by the monitoring hardware. Several techniques were used to do this:

- If source code is available, the application in question can be instrumented by adding logging statements to the relevant parts of the source code.
- Many applications have 'verbose' run-time configuration options which cause them to output extra information during execution—usually intended to aid in debugging. This debug information may be used to trace the application behaviour, without having to spend time on manually adding extra logging information, or if source code is not available.
- For very simple applications (in terms of network behaviour), it is sometimes sufficient for the application-level traces to consist simply of the start and stop times for the application, and any configuration options given at launch time.

Although additional application-level logging can affect application performance, it is not significant enough to alter the behaviour of the application when viewed at the network level, at least for the purposes of this work.

For the second case above, a tool was developed to make time-stamped recordings of the verbose logging output from an application that could later be correlated with the network-level timings. Correlation of these time-stamps requires the synchronisation of clocks on the network monitoring card and the hosts involved in running the applications. NTP [75] was used to synchronise system clocks to approximately millisecond accuracy. Greater accuracy (as offered by the use of a GPS time source) was not required.

The applications to be analysed (see Chapter 4) were driven in several ways:

Manual command-line runs or static batch scripts This technique was useful for interactive debugging of the monitoring software and for small, manually arranged, test runs.

Replay of network-level traces Rather than running the applications directly in the test environment, the machines were instead used to replay full-payload network packet traces onto the network. This can emulate a network containing thousands of hosts, provided that the source trace data is sufficiently diverse.

Synthetic network-level traces Simulating many thousands of clients by invoking the applications themselves is difficult without copious hardware. However, it was possible to test the monitoring system under different kinds of load

by programatically replicating network packet traces of a small number of transfers by using software to time shift packets and adjust IP addresses and port numbers. These were then replayed across the network.

The files used in the manual or script-driven runs of the application were a synthetically generated data set. This makes trace-level analysis easier since the data payload is known in advance. Sparse (zero-filled) Unix files of appropriate size were used because they take up little space on the source disk, and disk I/O is also negligible.

3.3 Trace Analysis

As mentioned above, network-level and application-level traces from particular applications performing file transfer tasks were gathered. The traces were manually inspected (using tools such as `tcpdump` and `ethereal`) to understand the basic operations of the protocol and identify characteristic features enabling them to be classified without resorting to well-known port number methods. In addition to the trace data, documentation about the protocols (where available) was used to guide the development of a plain text protocol analysis module.

These analyses were used to inform the design and implementation of the core of the real-time monitoring tool, bearing in mind the requirement of T3: "Such analysis can generate events early enough to enable dynamic traffic re-engineering."

Two main techniques were used for plain text control traffic analysis in real time: initial string matching for protocol identification, and full-payload application-protocol reconstruction for extraction of additional information about the application behaviour (see also Section 2.3). In addition to these, simple rate-based heuristics were employed for the identification of 'elephant'[1] flows.

The encrypted heuristic analysis used careful investigation of properties of the control and data traffic that can be observed at the network level to inform the development of a real-time analyser. It leverages both 'leaked' properties of the application protocol (i.e., information that has not been subject to encryption) and specific packet size/timing characteristics.

3.4 Testing and Evaluation

The system's event-reporting capabilities were regularly validated during development using trace files containing known behaviour. In order to evaluate the effectiveness of the system against unseen data, blind tests were carried out in which

1. The definition used here is of flows with a bandwidth above some threshold, as calculated over a pre-set time window. See also Section 2.3.1.

a third party was asked to run a series of the applications under study and keep application-level logs. They were later asked to correlate the output of the monitoring system with the application logs, in order to determine the system accuracy.

The heuristic analysis system was validated by using a set of test scripts that exercised a wide range of possible application uses, including comparing the performance between both relatively close and more distant hosts (within Glasgow University and between Glasgow and Lancaster).

During the execution of these tests, the time between an event being raised and the start of the bulk data flow to which it corresponds was measured.

Several metrics were identified for the system evaluation:

Accuracy: For plain text analysis, any positive output from the analyser should be 100% accurate in what it reports (no false positives), and the number of false negatives should be minimised. For heuristic analysis, false positives are a possibility, but false negatives should still be avoided.

Performance: In order for a real-time monitoring system to be useful, it must be capable of operating in real time. Therefore, it is necessary to evaluate the system under conditions of extreme load to determine the performance limits.

Maintainability: The analysers require application-protocol specific implementation. The costs involved in developing such an analyser must be evaluated.

3.5 Summary

The methodology described above allows the assertions from the Thesis Statement to be proven or contradicted. These are summarised below:

T1. **A real-time network monitoring tool can be built that performs full plain text analysis of Grid bulk data traffic at line rate.**
By using appropriate hardware and software techniques, a real-time monitoring system can be built. By achieving an understanding of the protocol operations of a selected set of Grid-style applications, protocol analysers can be developed within the real-time monitor.

T2. **This tool can be extended to perform heuristic analysis of encrypted or inaccessible payloads at line-rate.**
By careful examination and investigation of the network properties of an encrypted control protocol from a comprehensive set of source data, a heuristic technique can be developed to identify bulk data transfers in real time.

T3. **Such analysis can generate events early enough to enable dynamic traffic re-engineering.**

The combination of the above techniques, if properly implemented, can generate events in a sufficiently timely manner to enable re-engineering of the bulk data flows in the network.

Chapter 4 describes the Grid applications under study. Chapter 5 then describes the design and implementation of the real-time network monitoring infrastructure. Chapters 6 and 7 cover the plain text and heuristic traffic identification techniques, respectively, and Chapter 8 draws together the significant results of the work.

Chapter 4

Grid Bulk Transfer Applications

Modern scientific research is increasingly conducted with large collaborations, large shared data sets and large shared clusters of processors. Grid computing represents a wide range of application types and corresponding network usage. Indeed, network usage may be minimal in the case of compute-bound tasks (for example, parallel random simulations that require a small number of parameters and produce a similarly small amount of output). Applications that heavily utilise the local network but only use the wide-area network for transmitting summaries or aggregates also fall into this category. Wide-area, network-intensive, applications include bulk transfers, real-time multimedia, interactive visualisations and tightly coupled computations. Of these, bulk data transfers are the most relevant here because they tend to dominate the network load [49, 58].

4.1 Characteristics

In the following sections a number of bulk-transfer systems are examined and their characteristics as seen at the network level are considered. All the protocols under investigation are based on TCP connections. Each protocol is examined in terms of the following criteria:

Standard ports Does the protocol have standard port numbers allocated to it? Even though the real-time monitoring system is intended to identify traffic without resorting to well-known port identification, this information is still useful in the validation stages.

Reliability of port numbers How likely is it that traffic might be using a non-standard port number? Since end-users and system administrators are able to reconfigure the applications (e.g., via run-time options or by modifying source code), well-known ports cannot necessarily be relied upon.

Data separate from control Is the file content transferred over a separate TCP connection from the control connection? For some protocols this is always the case, for others it depends on run-time configuration.

Parallel data flows Does the protocol employ/support multiple parallel TCP flows for the transfer of data?

Third-party transfers Can files be transferred to/from hosts other than the pair involved in the control connection?

Plain text or encrypted Are control flows in the clear or encrypted? Are data flows in the clear or encrypted?

Signature payloads Are there simple signature strings that may be identified in packet payloads to identify the protocol?

A classifier/analyser has been developed for each of the applications considered here, and these are detailed in later chapters.

There are several applications used for making bulk data transfers in the context of the Grid. Indeed, the simplest of file transfer protocols like FTP and HTTP (described below) may be used, in addition to a range of specialised tools that meet the specific demands of high-bandwidth high-volume file transfers. The following sections describe a number of these applications, comparing their attributes and describing at a high level their effects on the network.

Before looking at the applications themselves in detail, some properties that are desirable for a Grid-level bulk transfer application are considered. Here, any project-specific concepts of the data to be transferred are abstracted away into the familiar concept of a file. Such a file may correspond to a real file on disk, or it could for example be a streamed representation of the contents of some kind of database or tertiary store.

A characteristic of the types of files transferred within the context of the Grid is their large size. Therefore, efficiency of data transfer and optimal utilisation of available network bandwidth are of prime importance. However, since Grid projects are by definition distributed across wide areas and perhaps also different administrative domains, a flexible authentication scheme is also important.

Beyond these two broad goals, a bulk transfer system may also be responsible for additional, higher-level, activities such as the management of user accounts and keeping track of locations of files and their possible replicas in the network. Simple protocols like HTTP and FTP deal with on-demand transfers of specific files as guided by some external process, whereas more complicated brokering systems such as the Storage Resource Broker (SRB) cover both account details and mapping of logical file names to physical storage locations.

The following sections give details on the chosen set of file transfer applications/protocols, which were selected to be representative of the kinds of tools used within the Grid community for transport of large files or making performance measurements [36, 95]. The set was extended to include the HTTP protocol because of its ubiquity and the particular challenges it poses for payload-level monitoring.

4.2 HTTP

As a ubiquitous and straightforward transport protocol, the Hypertext Transfer Protocol (HTTP) sees widespread use for general file transfer, mainly in the context of the world-wide web. Its popularity in this context often means that its well-known Transmission Control Protocol (TCP) port 80 traffic is treated specially (i.e., unfiltered) in environments where firewalling is in place. This in turn encourages application developers to make use of HTTP as a transport protocol when it would not otherwise be the best choice.

The HTTP protocol itself is very simple and is well specified in a series of Internet Engineering Task Force Request For Comments (RFC) [16, 44, 90]. Prior to HTTP/1.1, each file transfer corresponded exactly to the lifetime of a TCP connection. Therefore, the network behaviour could be simply described as follows:

1. Client opens connection to server
2. Client sends `GET` request with path to file to be retrieved
3. Server responds with header information
4. Server sends file to client
5. Client determines the end of the transfer when the server closes the TCP connection

The many, typically small, files encountered on common web pages meant that this protocol was inefficient, leading to the support for pipelined connection re-use found in HTTP/1.1. Here:

1. Client opens connection to server
2. Client sends `GET` request with path to file to be retrieved
3. Client may send more requests for additional files
4. Server responds with header, including enough information to describe the amount of data in the following file
5. Server sends file
6. Server repeats steps 4 and 5 for each incoming request

In summary, HTTP supports either precisely one or arbitrarily many file transfers per TCP connection, depending on protocol version. Control and bulk data traffic share the same connection, and there is no support for separating these.

Authentication in HTTP is generally accomplished by including an authentication token of some sort (e.g., plain text name/password pair or the result of a cryptographic computation with a challenge string) in the initial request header. If this is not present, the server will respond with an error reply and possibly a challenge to be used in a subsequent authentication attempt. The underlying systems that are used to manage the valid usernames and passwords are not specified.

A further development of the HTTP protocol adds encryption support. The protocol, known as HTTPS (using well-known TCP port 443), wraps the standard HTTP requests and responses in a Secure Sockets Layer (SSL) stream. SSL, de-

scribed in more detail in Section 4.8, uses public key cryptography and X.509 certificates to negotiate per-session encryption keys to be used in a stream cipher to secure the plain text protocol. X.509 certificates on the client side can be used for authentication (although this is relatively infrequently used), in addition to certificates provided by the server to prove its identify to the client.

The SSL connection is negotiated immediately after the TCP connection is opened and thereafter the TCP connection is used to transfer SSL frames which encapsulate messages from the underlying HTTP protocol. A consequence of this entire wrapping of the HTTP protocol in an encrypted stream is that bulk data transfers are subject to the computational overheads of encryption and decryption, which can be significant in the context of the high-bandwidth networks used by the Grid community. Whether or not bulk data encryption is necessary depends on the nature of the files being transferred. The implicit assumption here is that the authentication benefits and concealment of the paths and names of files being transferred is desired, since otherwise plain HTTP could be used.

HTTP Summary

Standard ports Ports 80 (HTTP) and 443 (HTTPS)

Reliability of port numbers High. Web traffic on non-standard ports is unlikely to be legitimate, given that web browsers assume the use of the standard ports.

Data separate from control No: all data is transferred over a single control/data stream.

Parallel data flows No.

Third-party transfers No.

Plain text or encrypted Depends on protocol version (HTTP or HTTPS).

Signature payloads For unencrypted HTTP, initial data is seen from the client of the form '`GET /path HTTP/1.x`'. For HTTPS, there is an initial TLS/SSL handshake, which by itself is not differentiable from other TLS/SSL based protocols. The direction and timing of encrypted messages may provide indicators that HTTP-over-SSL is being used.

4.3 FTP

The File Transfer Protocol (FTP) [89] predates HTTP by almost two decades. Its features grew out of requirements of computer networking in the 1970s and so has a particular feature set suited to the file transfer needs of that time. It was intended for interactive use, with the user connecting to an FTP server and issuing `GET` and `PUT` commands in order to retrieve and store files[1], as well as other familiar Unix-style file system operations (e..g, change directory, make directory, remove). This

1. In fact, the user-level `GET/PUT` commands map from a command-line interface to the lower-level FTP commands `RETR` and `STOR`.

```
<- 220 Dept. Computing Science FTP service - all activity is logged.
-> USER jp
<- 331 Please specify the password.
-> PASS ****
<- 230 Login successful.
-> SYST
<- 215 UNIX Type: L8
-> CWD /tmp
<- 250 Directory successfully changed.
-> PASV
<- 227 Entering Passive Mode (130,209,240,1,45,65)
-> LIST
<- 150 Here comes the directory listing.
<- 226 Directory send OK.
-> TYPE I
<- 200 Switching to Binary mode.
-> PASV
<- 227 Entering Passive Mode (130,209,240,1,245,66)
-> STOR sourcefile
<- 150 Ok to send data.
<- 226 File receive OK.
-> TYPE A
<- 200 Switching to ASCII mode.
-> PASV
<- 227 Entering Passive Mode (130,209,240,1,37,176)
-> LIST
<- 150 Here comes the directory listing.
<- 226 Directory send OK.
-> QUIT
<- 221 Goodbye.
```

Figure 4.1: Network trace of interactive FTP control session. Leading arrows have been added to show each message direction, with left-pointing arrows representing server-to-client messages, and vice-versa. The corresponding user input is shown in Figure 4.2.

two-way transfer capability contrasts with HTTP which is mainly used for server-client transfers (although HTTP does support transfers in either direction, it is uncommonly implemented and primarily used for WebDAV page edits from HTML editors). A transcript from a straightforward command-line FTP session is shown in Figure 4.2 and the corresponding network-level conversation in Figure 4.1.

Data transfers in FTP (for files and, notably, also directory listings) take place on a separate TCP connection from the control connection (which is on well-known TCP port 21). This allows flexibility in separation of the control and the data transfer processes, as well as supporting third-party file transfers. Although the FTP standards specify several schemes for the transfer of file content, only the simplest of these is commonly used. Here, the TCP socket is opened, the file contents are sent through the socket, then the socket is closed. End-of-file detection is intrinsically linked with the closing of the socket, with the unfortunate side-effect that unexpected socket errors can result in a silently truncated file[2].

The default in the FTP protocol is for the server to make a connection back to the client. In today's internet, these kinds of connections can be hindered by

2. This problem is also present in HTTP when a `Content-Size` header is not present. With both protocols problems can be detected when the file size is known.

```
$ ftp ftp.dcs.gla.ac.uk
Connected to ftp (130.209.240.1).
220 Dept. Computing Science FTP service - all activity is logged.
Name (ftp:jp):
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd /tmp
250 Directory successfully changed.
ftp> ls
227 Entering Passive Mode (130,209,240,1,45,65)
150 Here comes the directory listing.
drwx------    2 0        0            4096 Aug 29 08:13 ssh-lSyY6530
226 Directory send OK.
ftp> put sourcefile
local: sourcefile remote: sourcefile
227 Entering Passive Mode (130,209,240,1,245,66)
150 Ok to send data.
226 File receive OK.
39956 bytes sent in 0.00288 secs (1.4e+04 Kbytes/sec)
ftp> ls
227 Entering Passive Mode (130,209,240,1,37,176)
150 Here comes the directory listing.
-rw-r--r--    1 290      108         39956 Sep 07 19:57 sourcefile
drwx------    2 0        0            4096 Aug 29 08:13 ssh-lSyY6530
226 Directory send OK.
ftp> exit
221 Goodbye.
```

Figure 4.2: Transcript of interactive FTP session at the Unix command line. The corresponding network trace is shown in Figure 4.1.

the presence of firewalls and Network Address Translation (NAT) routers. As an alternative, the opposite connection direction can be negotiated where the client connects to a dynamically allocated port on the server, which is more likely to be supported by common firewall/NAT configurations. With either technique, arbitrary port numbers may be employed. This means that the data connection cannot easily be matched to the corresponding control connection.

Although FTP was originally used mainly interactively for command-line file transfer, involving interactively moving through the file system using familiar Unix-like commands (as depicted in Figure 4.2), it can also be used from scripts/applications through the use of the appropriate libraries. For example, Web browers generally have limited support for FTP, although the user experience isn't usually as good as with HTTP due to the additional complexities of the FTP protocol. The net result of non-interactive use of the FTP protocol is that the network-level characteristics of control protocol conversations are largely a function of application implementation specifics rather than user behaviour.

The auxiliary data connections are not authenticated, so there is scope for an attacker to connect in place of the real user. Properly implemented clients and servers would safeguard against this by checking Internet Protocol (IP) addresses of peers before exchanging data, however modern security practices dictate that by itself this is insufficient. Similarly, FTP uses a simple plain text username and pass-

word combination for login purposes. As with HTTP, there are standards-based extensions to to support more modern authentication mechanisms. RFC 2228 [52] specifies FTP extensions for alternative authentication schemes, encryption of control conversations, and optional authentication and/or encryption of data streams.

The extensions manage the encrypted exchange of control traffic by passing the plain text messages through an encryption and/or integrity protection algorithm and then transmitting the Base64 encoded result in one of a series of new FTP commands. These are `ADAT`, `MIC`, `CONF` and `ENC` (for exchange of authentication data, integrity protected commands, confidentiality protected commands, and privacy protected commands, respectively). The actual security algorithms to be used are negotiated by another command, `AUTH`, for which the Generic Security Services Application Program Interface (GSSAPI) is described. As the name suggests, GSSAPI is a generic API for performing secure message exchange. There can be many concrete implementations of the API, and it is up to the client and server using FTP with GSSAPI to use the same implementation. An example of a concrete implementation is a GSSAPI wrapper for Kerberos. GridFTP, described in Section 4.4 specifies another.

FTP Summary

Standard ports Port 21 (control). Port 20 is sometimes used for one side of a data connection.

Reliability of port numbers High. FTP traffic on non-standard ports is unlikely to be legitimate, given that clients (including web browsers) assume the use of the standard ports.

Data separate from control Yes.

Parallel data flows No.

Third-party transfers Protocol support, but rarely used.

Plain text or encrypted Plain text.

Signature payloads `220 hello` message from server. `USER` and `PASS` commands and their respective responses.

4.4 GridFTP

GridFTP [8] is a specialisation of the extended FTP protocol from RFC 2228. It implements the security extensions by using a GSSAPI library wrapping the Grid Security Infrastructure (GSI). GSI is a public key based security system which supports mutual authentication with digital certificates, credential delegation and single sign on. These features are vital for Grid projects in order to effectively support the complexities of managing geographically distributed computer systems that fall within different administrative domains. The low-level encryption algorithm

implementation in GSI (compared with, for example, the higher-level certificate management policies and credential delegation mechanisms) is in fact simply SSL/TLS.

In addition to the specific security mechanisms in place, GridFTP extends standard FTP by providing for multiple parallel TCP streams for the transfer of files. As discussed in Section 2.1.1, multiple streams can be used to better utilise the available network bandwidth. The presence of multiple parallel streams is a common characteristic of bulk transfer applications, as also exhibited by SRB and BBFTP (Sections 4.5 and 4.6).

Figure 4.3 on page 45 is a transcript of a sample GridFTP control session. The left-hand column shows the truncated representations of the textual lines sent and received on the connection. The middle column of numbers in square brackets indicates the original character count for the truncated lines. The leading arrows indicate the direction of the message and are not part of the original data. A left-pointing arrow represents data from server to client, and vice-versa. The GridFTP session begins with the selection of the GSSAPI security protocol (line 2). Lines 4–11 negotiate the encryption algorithms to use and exchange X.509 certificates. The square-bracketed numbers indicate the size of these exchanges, which are much larger than anything that follows. Beginning at line 12, the right hand column shows the plain text corresponding to the encrypted data in the left column. Since certificates have been exchanged, the `USER` and `PASS` commands at lines 12 and 14 do not carry user-specific parameters. The client then proceeds to interrogate the server to find out its capabilities (via the `SITE HELP` and `FEAT`) commands. The multi-line responses to these commands are delivered in several encrypted lines (17–23 and 25–35).

At this point it's interesting to note that the version of the Globus GridFTP server used in the test leaks a small amount of 'confidential' information in the encrypted text due to an invalid assumption in the source code. This can be seen on lines 18–22 and 26–34 where the character immediately following the number `631` in the left hand column should be a dash (-) but instead corresponds to the fourth character of the plain text. This type of leak could be used to validate inferences made by the heuristic analyser.

Continuing with the transcript walk-through, line 38 sees the client requesting the size of the file to be transferred (only 1.5 Kbytes) before setting the transfer buffer size to 1 Mbytes (line 40). Line 43 indicates the IP address and port number on the server for the client to connect back to. Following this, the `RETR` command is issued and the transfer takes place on the auxiliary TCP connection. Once complete, the control connection sees the 'Transfer complete' response on line 46 and then the goodbye messages at the end of the session.

Although GridFTP has a well-known TCP port number (2811) assigned by the Internet Assigned Numbers Authority (IANA), site administrators are free to

```
<- 220 tim GridFTP Serv [ 109] 220 tim GridFTP Server 1.17 CAS/SAML enabled GSSAPI
-> AUTH GSSAPI          [  11] AUTH GSSAPI
<- 334 Using authentica [  54] 334 Using authentication type GSSAPI; ADAT must fol
-> ADAT FgMAAGEBAABdAwB [ 141]
<- 335 ADAT=FgMAAEoCAAB [1881]
-> ADAT FgMABuQLAAbgAAb [2749]
<- 335 ADAT=FAMAAAEBFgM [ 109]
-> ADAT FwMAACD++DJ4iF0 [  57]
<- 335 ADAT=FwMAATCOfnp [ 421]
-> ADAT FwMAAjBruxUZ9CA [ 761]
<- 235 GSSAPI Authentic [  35] 235 GSSAPI Authentication succeeded
-> ENC FwMAADCyMbN+ZRnH [  76] USER :globus-mapping:
<- 631 FwMAAID+J5F5HEqV [ 184] 331 GSSAPI user /O=Grid/OU=GlobusTest/OU=simpleCA-g
-> ENC FwMAACD8dDV4ARCU [  56] PASS
<- 631 FwMAADBZ0AsMcxPO [  76] 230 User jon logged in.
-> ENC FwMAACDAhe/D7q7e [  56] SITE HELP
<- 631-FwMAAGBk0GFJbCxw [ 140] 214-The following SITE commands are recognized (* =
<- 631UFwMAAGAoo6489QgL [ 140]    UMASK           GPASS           ALIAS
<- 631IFwMAAFCQ2NL0aR7b [ 120]    IDLE            NEWER           CDPATH
<- 631CFwMAAFB4U7lxP51h [ 120]    CHMOD           MINFO           GROUPS
<- 631HFwMAAFDfsfoKugeJ [ 120]    HELP            INDEX           CHECKMETHOD
<- 631GFwMAAFBX+coRHNqZ [ 120]    GROUP           EXEC            CHECKSUM
<- 631 FwMAAEDB0KZfSkrz [  96] 214 Direct comments to ftp-bugs@tim.
-> ENC FwMAACDIJU6HbJAM [  56] FEAT
<- 631-FwMAADBPvp5a2tbA [  76] 211-Extensions supported:
<- 631SFwMAADAeASL0UybR [  76]  REST STREAM
<- 631TFwMAACAFN46GWb/z [  56]  ESTO
<- 631EFwMAACC1HA0YAyTR [  56]  ERET
<- 631TFwMAACAsagSQIMAr [  56]  MDTM
<- 631SFwMAAGDyIYuvZRh8 [ 140]  MLST Type*;Size*;Modify*;Perm*;Charset*;UNIX.mode*
<- 631ZFwMAACBUdyQ96SwR [  56]  SIZE
<- 631SFwMAACCYjsKkGWpp [  56]  CKSM
<- 631RFwMAACCtaYzKOS3t [  56]  PARALLEL
<- 631AFwMAACA4oeCotGeB [  56]  DCAU
<- 631 FwMAACCm8qMLZvSM [  56] 211 END
-> ENC FwMAACD9mR0JDlTO [  56] TYPE I
<- 631 FwMAADDnIwMQQn9V [  76] 200 Type set to I.
-> ENC FwMAADDzQK5htDbZ [  76] SIZE /tmp/source-file
<- 631 FwMAACAZg/CkCo/u [  56] 213 1586
-> ENC FwMAADCz26In5BvS [  76] PBSZ 1048576
<- 631 FwMAADBPrKAZ/T9T [  76] 200 PBSZ=1048576
-> ENC FwMAACAba5QvAk8M [  56] PASV
<- 631 FwMAAFDch2m4PFgN [ 120] 227 Entering Passive Mode (192,168,1,23,130,177)
-> ENC FwMAADC8pctm//jb [  76] RETR /tmp/source-file
<- 631 FwMAAEA/E9grU3HU [  96] 150 Opening BINARY mode data connection.
<- 631 FwMAADDnpTivqpYq [  76] 226 Transfer complete.
-> ENC FwMAACAEq8al6Rz+ [  56] QUIT
<- 631-FwMAAFDkw9QIxxQn [ 120] 221-You have transferred 1586 bytes in 1 files.
<- 631-FwMAAGASPUTbbfUv [ 140] 221-Total traffic for this session was 9474 bytes i
<- 631-FwMAAFBdiQ0mQor7 [ 120] 221-Thank you for using the FTP service on tim.
<- 631 FwMAADDFbIYt7LD7 [  76] 221 Goodbye.
```

Figure 4.3: Encrypted GridFTP control connection (left, heavily truncated) alongside plain text (right, less truncated). Leading arrows indicate direction of communication, where a left-pointing arrow represents data from server to client, and vice-versa. The middle column of numbers in square brackets indicates the total number of characters before truncation in each line of the original conversation.

run servers on ports of their choosing, perhaps in order to work with local firewalling restrictions. Therefore, it is desirable to be able to identify GridFTP traffic as such by inspecting packet contents or making statistical inferences based on the higher-level traffic properties. GridFTP (using GSSAPI-GSI security) has a number of characteristic features that are likely to be useful in traffic identification. For example: the GridFTP hello message from the server; the GSSAPI authentication exchange; and certificates issued by known Grid certificate authorities.

GridFTP Summary

Standard ports Port 2811 (control).

Reliability of port numbers Low. Although port 2811 is allocated by IANA, administrators may choose different ports according to local policy. Use of non-standards ports is not a user-interface problem because GridFTP URLs will usually be generated, stored and manipulated programatically.

Data separate from control Yes.

Parallel data flows Supported.

Third-party transfers Supported.

Plain text or encrypted Semi-plain text, encrypted commands.

Signature payloads `220 hostname GridFTP` initial message from server. GSSAPI authentication negotiation. Certificate exchange containing Grid-issued certificates.

4.5 Storage Resource Broker

The Storage Resource Broker [14] is a much more complicated system than the protocols described above. Rather than just being a file transfer application, it provides a virtual file system (represented logically in an SQL database) that may be accessed through the SRB protocol. The virtual file system provides mappings from logical file names to physical storage locations, which may in turn be comprised of different types of data resources. On-line disk storage is the most conventional of these, but projects with huge storage requirements and compatible access patterns may use a hierarchical storage system, such as a layering of archival mass storage (e.g., magnetic tape) and normal disks for short-term access.

In addition to the basic virtual file system front-end, SRB aids in management of file replicas across distributed storage resources. Each storage system (disk, tape, etc) is attached to a host which runs the SRB server software. Each SRB server maintains an association with the Metadata Catalog (MCAT) which manages the logical file name mappings and replica catalog.

The following paragraphs describe in detail the different components of an SRB installation, starting with the central component—the MCAT.

MCAT The Metadata Catalog uses an SQL database backend to store username-password pairs, descriptions of available physical storage systems and their corresponding SRB server hosts, and the logical file system directory tree and pointers to the corresponding physical storage locations. An MCAT is a normal SRB server with additional functionality that is selected at software compile time.

Server Each server contacts its statically configured MCAT to learn about the physical resources present on each system. This means that no resource-specific configuration is stored on the individual servers—everything is stored centrally in the MCAT. Clients connecting to a server are authenticated by passing on their credentials to the MCAT for verification. Servers may connect to each other to perform file transfers, in which case a super-user account is used to implement the credential delegation (in effect, the servers and MCAT form a network of trust).

Client Tools A set of command-line client tools are provided that mimic the usage of standard Unix tools. For example: `Sls`, `Scd`, `Sput` and `Sget`. In addition, a C and Java library are available for making direct connections to the SRB system from end-user code.

The SRB client is configured through the use of Unix dot-files in a user's home directory. These files contain the address of the SRB server to contact, as well as the user name and password to be used for the connection. An SRB client can be configured to connect to any SRB server in the network, since all authentication is eventually proxied to the MCAT.

Communication between clients and servers uses a custom Remote Procedure Call (RPC) protocol, which operates in the default configuration in plain text. After an initial authentication handshake is performed, the client issues synchronous RPC calls. These largely mimic the standard Unix file operations (open, close, read, write, etc.) and therefore the SRB client library is a thin wrapper around the RPC protocol. A consequence of this architecture is that large file transfers involve interleaving control messages and data in the same TCP connection. For example, by default file transfers are performed using a transfer buffer size of 4 Mb, which means that there will be regular re-issues of read/write RPCs throughout the transfer, carrying with them an associated loss in throughput. Furthermore, since data is transferred over the single control connection, any file transfers that come from a server other than the one to which a client has connected will be proxied through the first-hop server. This is an inefficient use of network bandwidth.

Extensions to the SRB protocol have added additional procedures that negotiate auxiliary TCP connections to be used for the transfer of data. With this scenario, transfers can be made directly from the client application to the server managing the data resource, provided firewall restrictions allow it.

Figure 4.4 is a block diagram showing the different relationships between

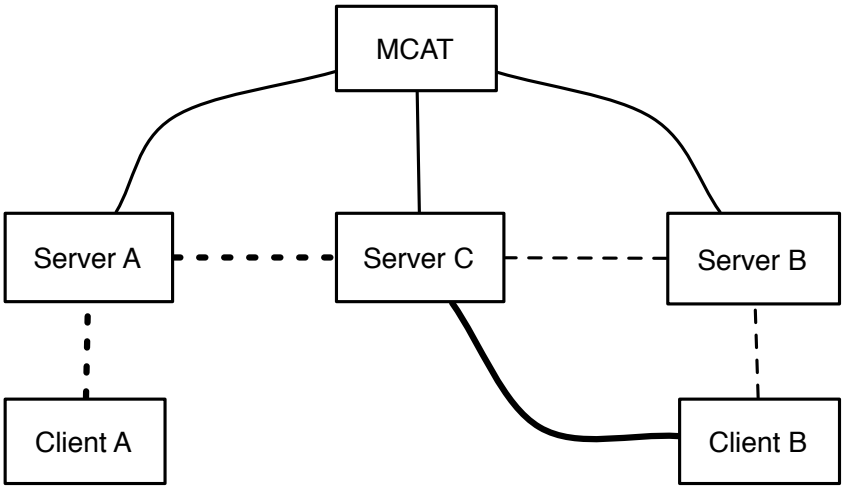


Figure 4.4: Block diagram showing relationship between different components of an SRB installation.

MCAT, Server and Client. Each Server, which provides access to one or more data resources, connects to the MCAT for local resource configuration, logical file name mapping and user authentication (thin solid line). Clients connect to their designated Server (dashed lines). Depending on Client configuration (whether auxiliary TCP streams are being used), data is transferred over the control connection (thick dashed line) or over one or more dedicated data connections (thick solid line). In the former case, data to be transferred is proxied through the designated Server. In the latter, a direct connection is established between Client and the Server responsible for the data resource required.

At configuration/compile time, the SRB software can be configured to make use of an encryption library to wrap the plain RPC protocol. It supports GSSAPI (and therefore GSI) as well as a custom encryption library known as SEA (SDSC Encryption/Authentication). While this makes the RPC protocol inaccessible by direct payload inspection in a network monitor, it brings it to the same level as GridFTP in terms of identification and statistical inferences.

SRB Summary

Standard ports Port 5544 for broker connection. Other ports allocated dynamically.

Reliability of port numbers Low. Administrators may choose non-standard ports according to local policy. Use of non-standards ports is not a user-interface problem because the port number must always be specific when configuring the SRB client software.

Data separate from control Depends on run-time configuration option.

Parallel data flows Supported.

Third-party transfers Yes (for example, the data transfer between Client B and

Server C from Figure 4.4 wherein Client B's control connection is with Server B).

Plain text or encrypted Both plain text and encrypted are supported, depending on compile-time configuration.

Signature payloads Broker client sends `START SRB` as a null-terminated C-style string. Server replies with port number as a 32-bit integer. This is independent of any configured security mechanisms (e.g., GSSAPI).

4.6 BBFTP

BBFTP [41] was developed to support the BaBar project, transferring huge data files between the Stanford Linear Accelerator Center (SLAC) and the the In2p3 Computing Center in Lyon, France. It is a very light-weight system that requires minimal configuration at client and server.

The protocol is implemented solely in the `bbftp` command line tool and corresponding `bbftpd` server. The client takes a single instruction or batch file of instructions for files to transfer. It has support for on-the-fly compression of data files and was explicitly designed to use parallel TCP streams to gain most effective use of available network bandwidth. TCP connections may be made from client to server or server to client, depending on run-time configuration options.

Authentication may be accomplished one of several ways, determined at compile time:

1. Simple user name and password, verified on server against Unix password file.
2. AFS/Kerberos tickets.
3. GSI via GSSAPI.
4. Pre-authentication as the user running the `bbftpd` server process.

For option 1, name and password may be passed in clear text, or encrypted using a public-key scheme using the OpenSSL library. Regardless of which authentication mechanism is employed, the rest of the control protocol is not encrypted. In option 4, the server process is usually started from a remote shell connection, for example via Secure Shell (SSH). In this case, the control protocol conversation takes place over the server process standard input/output files and therefore the observable network traffic is the SSH-encrypted modulation of the basic BBFTP protocol.

The BBFTP protocol conversation after authentication is a conceptually simple request-response scheme that exchanges file paths and port numbers to be used for the auxiliary parallel TCP connections. The actual implementation of the protocol, however, is somewhat irregular and poorly documented. Development of a reliable network analyser requires inspection of the source code to identify deviations from the regular structure of the request-response scheme. Furthermore, no

Field	Meaning
`flags`	Bit field.
`numThreads`	Number of parallel TCP streams to use.
`port`	TCP port to connect back to.
`bufferLen`	Application-level read/write buffer size.
`winBand`	TCP window size (or 0 for default).
`amount`	Positive indicates number of bytes. Negative indicates time in milliseconds.

Table 4.1: Fields in the `iperf` client header message. Each entry is a 32-bit integer. Since there are six fields the total message size is 24 bytes.

special identification strings are used in the initial exchange to ensure that both sides of the connection are using the same protocol. This means that client/server negotiation of different protocol versions is dependent on probing for failure responses to new-version messages, and that identification of the protocol from an external analyser requires knowledge of the peculiarities of this process.

BBFTP Summary

Standard ports Port 5021 default control connection. Some data connections established using 5020 as source port.

Reliability of port numbers Medium. Administrators may choose non-standard ports according to local policy. Although using a non-default port requires extra configuration options, automated invocations of BBFTP can easily incorporate non-standard configurations.

Data separate from control Yes.

Parallel data flows Yes.

Third-party transfers No.

Plain text or encrypted Plain text apart from authentication process, unless control protocol transported over a remote shell.

Signature payloads No explicit 'magic' protocol identification strings. However, the initial authentication exchange messages follow a basic RPC scheme that can be identified.

4.7 iperf

The `iperf` [101] tool is used to evaluate network throughput using TCP or UDP. The number of parallel TCP connections and the TCP buffer sizes are configurable.

Although `iperf` is not strictly a Grid bulk data transfer application, it is widely used to determine the throughput of a network path and identify problems [37]. A typical configuration is to run on a regular basis (e.g., hourly) to evaluate performance of important network links over time. The network load resulting

from such usage can be significant, and as such it is useful to be able to identify and quantify `iperf` usage.

The `iperf` server and client are configured at the command line, and have a default TCP port on which the initial connection is established. Bandwidth tests are conducted over this connection, and over further connections if parallel mode is requested. Data transfer may be in either (or both) directions, depending on run-time configuration options. In the default single-direction mode, the client makes one or more connections to the fixed server port and transmits data. In 'dual' or 'tradeoff' modes, the server connects back to the client *on the same fixed port number* to transfer data. 'Dual' mode makes simultaneous transfers in both directions, whereas 'tradeoff' mode sends data from client to server and then from server to client.

There are no well-defined 'magic strings' or other obviously recognisable protocol features in the network-level conversation. Upon connection from client to server (but not server to client), a parameter block containing the number of parallel connections, buffer size, window size and test duration is exchanged. A heuristic can be used to identify TCP connections as `iperf` by attempting to interpret initial stream contents of every TCP stream as one of these parameter blocks. If the resulting values make sense, further processing on the connection can take place. For example, Table 4.1 shows the fields present in the `iperf` client message header. Each field is a 32-bit integer but the `numThreads` and `port` fields, for example, will never contain a value outside the range of a 16-bit integer. This adds a constraint for recognising this initial control block. A byte-level regular-expression style filter can be employed to make a first-cut classification that eliminates any initial contents that are plainly not `iperf`; implementation details are described further in Sections 5.3.4 and 5.4.3.

iperf Summary

Standard ports Port 5001 default.

Reliability of port numbers Medium. Administrators may choose non-standard ports according to local policy, or perhaps to investigate network performance problems. For example, an operator may employ packet shaping techniques that depend on TCP port numbers.

Data separate from control No.

Parallel data flows Yes.

Third-party transfers No.

Plain text or encrypted Plain.

Signature payloads No. Heuristics required.

4.8 TLS/SSL

Although TLS/SSL is not a bulk data transport protocol in itself, its use in some of the above applications, as well as in the heuristic analyser described in Chapter 7, merits separate treatment. The protocol provides privacy and data integrity between two communicating processes. It specifies two layers: the Handshake Protocol and the Record Protocol. The Handshake Protocol is layered on top of the Record Protocol, which in turn is layered on top of some reliable transport protocol—usually TCP. At the start of a connection, the Handshake Protocol is used to exchange encryption keys and verify identities. Thereafter, the application is free to transport its own data in encrypted payloads encapsulated within the Record Protocol.

The Record Protocol itself consists of a stream of type-, version- and length-prefixed messages. These elements are transmitted in the clear, and enable the framing of the Record Protocol to be identified by monitoring the TCP-level communication.

The identity verification step mentioned above uses X.509 certificates [29], which contain the name of the subject (amongst other attributes) and a reference to the certificate of some other entity that attests the accuracy of (i.e., signs) the information contained in the certificate. Certificates used by web servers, for example, are typically signed by one of the major certificate authorities, which are in turn trusted by major web browsers. The Grid community, on the other hand, establishes its own chain of certificate authorities, over which the organisations involved have control.

4.9 Summary

The preceding sections have described the behaviour, at a high level, of several bulk data transfer applications, as might be expected to be used by Grid-style projects. Some of these are well-established internet protocols (such as HTTP and FTP), whereas others (SRB, BBFTP, iperf) are communications systems whose behaviour is dictated almost entirely by the specifics of a single implementation. GridFTP, unusually, is a standards-based protocol that has evolved out of the needs of the Grid community. It builds upon the framework of extended FTP (RFC 2228) and provides one of the few implementations of the abstract encryption/authentication scheme there.

The protocols have been characterised by the possible behaviours at the network level for the higher-level file transfer operations invoked by the user (or controlling software entity). The simplest of these characterisations is the use of standard TCP port numbers—although in every case the software may be reconfigured

	HTTP	**FTP**	**GridFTP**	**SRB**	**BBFTP**	**iperf**
Standard ports	80	21[1]	2811[1]	5544[1]	5021[1]	5001
Reliability of ports	High	High	Low	Low	Med.	Med.
Separate data	No	Yes	Yes	Maybe	Yes	Both
Parallel data flows	No	No	Yes	Yes	Yes	Yes
Third-party transfers	No	Yes	Yes	Yes	No	No
Plain/encrypted	HTTPS[2]	No	Yes	Maybe	SSH[2]	No
Signature control	Yes	Yes	Yes	Yes	No	No
Signature data	N/A	No	Yes[3]	No	No	No

1. Control only 2. Optional 3. With data channel authentication

Table 4.2: Summary of bulk transfer application details.

to use site-specific ranges as dictated by firewall restrictions or policy. Table 4.2 summarises the properties of each of these protocols.

Chapter 5

Real-Time Application Protocol Analyser

In order to achieve the goal of recognising and reporting on Grid application traffic, a real-time application protocol analyser was designed and implemented. This chapter describes the system.

Since the applications of interest are all TCP-based, it is sufficient to focus on the reassembly of TCP flows as the main monitoring technology. Reassembled flows can be handed off to protocol analysis modules (specific implementations of which are described in Chapter 6) and from there events may be generated for delivery to a management system.

This chapter is structured as follows. After describing the requirements, a prototype developed using Bro (see Section 2.4.3) is discussed. The limitations encountered therein, along with the initial requirements then drove the design of the system in the following sections. Finally, some specifics of the implementation are dealt with, followed by an evaluation of the system performance.

5.1 Requirements

The core requirements are as follows:

R1. Monitoring at full duplex Gigabit Ethernet line rate.
R2. Reassembly of all TCP flows.
R3. Content-based application protocol identification rather than use of well-known ports.
R4. Framework for easily developing concurrent full-payload protocol analyser modules.
R5. Simple event reporting mechanism for integration with a management system.

The first requirement follows immediately from the first assertion of the thesis statement, T1 on page 12. Full-duplex Gigabit Ethernet implies a maximum bandwidth of 2 Gbits/s as seen by a network monitor observing both flow directions. The costs (in terms of CPU time) of traffic analysis at this rate depend on two main

factors: the proportion of flows for which protocol analysers are required and the distribution of packet sizes. If many flows are being actively monitored, demand for CPU time will increase. Assuming that most traffic on the link is bulk (and therefore not subject to content processing), small packets will lead to a greater number of packets in a given time and therefore CPU time will increase correspondingly.

Full reassembly of all TCP flows (R2) is vital for the accurate analysis of the application-level protocols. It is assumed that the flows of interest are routed symmetrically with respect to the monitoring point. If this were not the case, it would not be possible to reconstruct both sides of the TCP conversation since a flow is not guaranteed to be seen by the monitor.

Instead of depending on well-known ports for initial flow classification, requirement R3 dictates that flow content be used to classify flows. Although this is computationally more expensive, it is a much more powerful monitoring technique than simple port-based identification, and becomes necessary when end users or site network administrators adjust run-time or compile-time port number configurations, perhaps to accommodate firewall/policy restrictions.

Since the approach for analysing the control protocols of interest is to fully reconstruct application-level message exchanges, it is necessary for the real-time monitoring system to support multiple concurrent analysis modules. For the system to be truly useful, the development costs associated with the creation of an analysis module must be minimised (R4).

Finally (R5), the system must be able to deliver events generated by the analyser modules to a management system. A simple mechanism is sufficient to demonstrate the effectiveness of the monitor. Subsequent integration with a management system should be straightforward.

In summary, the requirements are for a line-rate Gigabit Ethernet monitoring system with the ability to classify TCP streams, based on initial content, before handing off subsequent stream processing (if needed) to a protocol-specific analyser module. These modules deliver events that classify control and bulk flows according to protocol.

5.2 Bro Prototype

The Bro network intrusion detection system was used as a framework for initial prototyping of the system. As described in Section 2.4.3, Bro is intended to be a Gigabit Ethernet network monitor that supports accurate reassembly of TCP flows. Application-protocol specific analysers written in C++ are responsible for initial analysis of a particular flow and a separate policy is implemented using a custom scripting language.

These features of Gigabit Ethernet monitoring and TCP flow reassembly precisely fit requirements R1 and R2. A simple C++ protocol analyser for the SRB protocol (see Section 4.5) was implemented using Bro's framework. A limitation was encountered at this point with respect to the handling of the SRB broker connection, which negotiates an ephemeral port to be used for the main SRB session. Since Bro's `libpcap`-based packet filter is statically defined to include just the well-known ports corresponding to the enabled protocol analysers, it was not possible for the SRB analyser to see forthcoming traffic on the negotiated port.

Proper support for this would require the ability to modify the incoming packet filter on the fly, in order to be able to process packets on the newly identified port (which would not previously have been included in the packet filter rules). Unfortunately, even though Bro could be augmented to support live modification of the packet filter, it is likely that packets of interest would be lost in the time window between recognising the SRB broker protocol and the new in-kernel packet filter taking effect.

To work around this problem, the Bro packet filter was set to capture all TCP traffic, irrespective of port number. Unfortunately, when running in this configuration the system became overwhelmed by bulk traffic on the link due to the overheads of TCP reassembly. Although this could be mitigated to some extent by avoiding reassembly for flows that are known to be bulk data, the case of mixed control and data (where both types of traffic within one TCP connection need to be efficiently interpreted) could not be handled.

For these reasons, a stand-alone system optimised for application protocol monitoring of bulk data transfers was devised. Building the SRB analyser in Bro was instructive and the analyser implementation has been ported without major modifications to work in the system described in the rest of this chapter.

5.3 Design

This section covers the key design decisions made during the development of the real-time analyser. Given the experiences of using Bro, it was clear that an efficient mechanism was required to manage TCP flow reassembly of different types of flows (control vs. data). Furthermore, care would be necessary to reduce demands on the memory bus and CPU in general so that the analyser could achieve the line-rate of Gigabit Ethernet (as specified in R1).

The main architectural difference from Bro is the decision to exploit a circular buffer packet capture interface and to assume the availability of dedicated network monitoring hardware that provides this interface. The advantages of this architecture will be described in detail in the next section.

Several assumptions and simplifications were made to reduce the scope of the project. These are listed below:

1. *TCP only (no UDP)*
   The applications under study are only using TCP. The analyser could be extended in the future to support other IP protocols via auxiliary protocol reassembly modules.
2. *IPv4 and IPv6*
   Although TCP transport is assumed, both IPv4 and IPv6 are explicitly supported.
3. *No IP fragment handling*
   Properly configured end hosts should be able to negotiate the appropriate maximum segment size (MSS) for the end-to-end path. Indeed, this is necessary in order to obtain maximum throughput (as desired by bulk transport protocols). If evidence later suggests that IP fragments are more prevalent than assumed here, additional reassembly support could be added.
4. *Studied applications are not used maliciously*
   Systems like Bro take steps to correctly process specially crafted packets that are designed by attackers to bypass intrusion detection. The assumption here is that traffic resulting from the protocols under study is not modified in this way.
   Related to this is the issue of dealing with malicious traffic that is not covered by the above assumption. For example, there may be denial of service attacks crossing the monitoring point. Ideally, the monitor should be able to cope gracefully with such traffic. Experimental evidence in Section 5.5.1 shows that the monitor copes with SYN flooding, a common form of denial of service attack.

Throughout the design and implementation process, a number of design principles were kept in mind to ensure that the completed system would operate as efficiently as possible:

1. *Minimise memory accesses*
   As little data as possible from each packet should be inspected and, similarly, packet data should be copied as little as possible, both to minimise CPU load and also to reduce contention on the PCI bus, which is shared by the packet capture card and the CPU. If the bus is too busy then the card may not be able to claim sufficient memory bandwidth to store incoming packets. In effect, the goal is to avoid introducing any per-byte overheads into the analyser that would become a bottleneck for scalability.
2. *Minimise heap allocation*
   Heap-based memory allocation/freeing can be time consuming and causes heap fragmentation. For a long-running monitoring system, this is a concern.
3. *Process packets as soon as possible*
   If packets are not processed when they arrive, they must be buffered. Buffer-

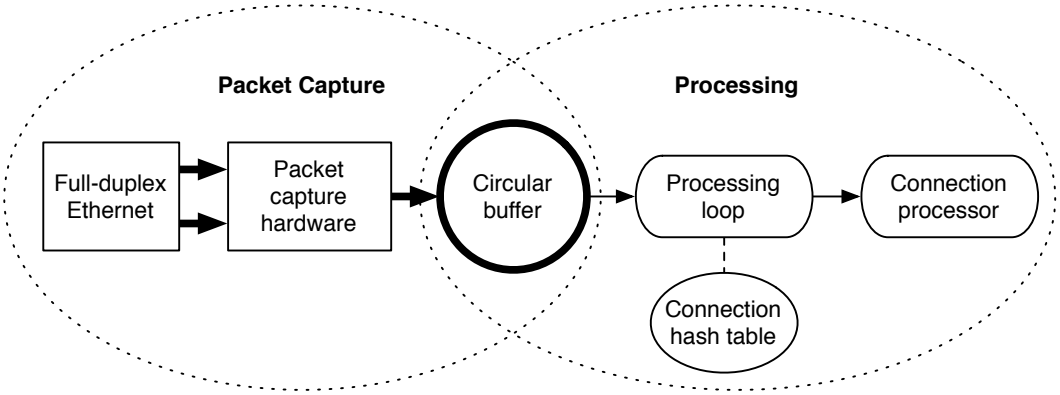


Figure 5.1: Block design of real-time analyser, showing the circular buffer link between the packet capture hardware and the packet processing software.

ing costs time and memory, and should be avoided where possible.

4. *Single-threaded, data-driven structure*
   To simplify matters, the system is intended to run on a single processor, with processing driven by the arrival of network packets. Potential for extension to multiprocessor machines or multiple monitoring machines, operating in unison, is future work (see Section 5.5.4).

5.3.1 Circular Buffer Capture Interface

The monitoring system is built around a main loop driven by the arrival of captured packets in a circular buffer. Figure 5.1 is a block diagram showing the high-level structure.

The capture card stores incoming packets into a traditional circular buffer of several hundred megabytes. In parallel, the application processes the packets in the buffer, and uses a hash table of TCP connections in order to track the per-connection state. Once a connection has been identified, further processing is handed off to another module.

The basic circular buffer arrangement is shown in Figure 5.2. The capture hardware advances the 'Write Pointer' and will fill up to the 'Application Acknowledged Pointer'. As the application processes the packets, any non-TCP packets are immediately skipped. A canonical[1] quadruple consisting of (source port, source address, destination port, destination address) is formed and looked up in the connection hash table. If a lookup fails, additional lookups are performed against the quadruples (source port, source address, 0, 0) and (destination port, destination address, 0, 0). These correspond to provisional connections that may have been installed by a protocol analyser that is expecting additional related connections (for example, an FTP data connection whose destination IP/port is known but the

1. The source and destination pairs are ordered in the connection table according to their numeric value.

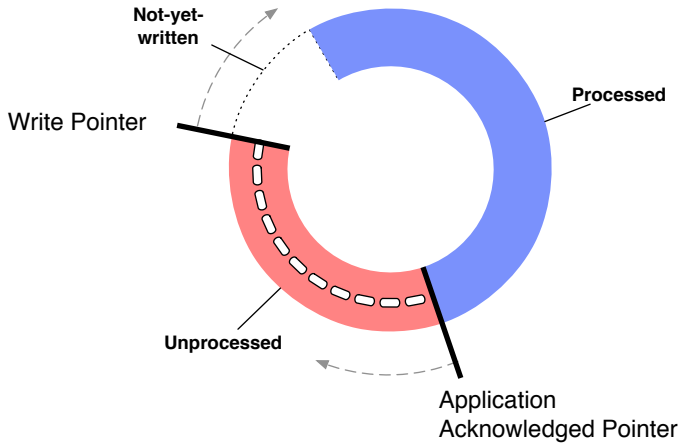


Figure 5.2: Circular capture buffer, showing the head of the circular queue (the "Write Pointer") and the tail of the queue (the "Application Acknowledged Pointer").

source IP/port is unknown).

If all these lookups fail, a new connection is allocated and installed in the table. Either way, the incoming packet is eventually dispatched to the connection handler.

Aside from packet processing, the main loop also handles timers. These are set up by the connection handler or protocol handler, and are mainly used to expire connection records after inactivity or once a connection has been closed.

5.3.2 TCP Reassembly

By handing off packet capture to dedicated hardware, the CPU is free to take care of analysis tasks rather than process interrupts and take care of buffer management, as would normally be expected from the use of a traditional Ethernet card as a capture source. In addition to these performance improvements, the circular buffer can be further exploited for efficient real-time TCP flow reassembly.

Out-of-order TCP segments are treated specially: they are left in the capture buffer rather than being copied onto the heap. A per-flow[2] linked list of these *retained packets* is maintained. In addition, a global linked list is constructed to allow the main event loop to keep track of the lower limit in the circular buffer to which the capture hardware is allowed to write. The first entry in this list always points to the closest packet to the capture write pointer. The circular buffer arrangement incorporating the retained packet scheme is shown in Figure 5.3.

The linked list data structure in each retained packet overwrites the capture record header and some of the Ethernet header. Since only the IP protocol and layers above it are of interest, this is not a concern. In fact, by the time a packet has become eligible for being retained, the IP header is no longer of interest so the space occupied by it could be used for additional retained-packet book-keeping.

2. Each connection record is composed of two flow records, one for each direction.

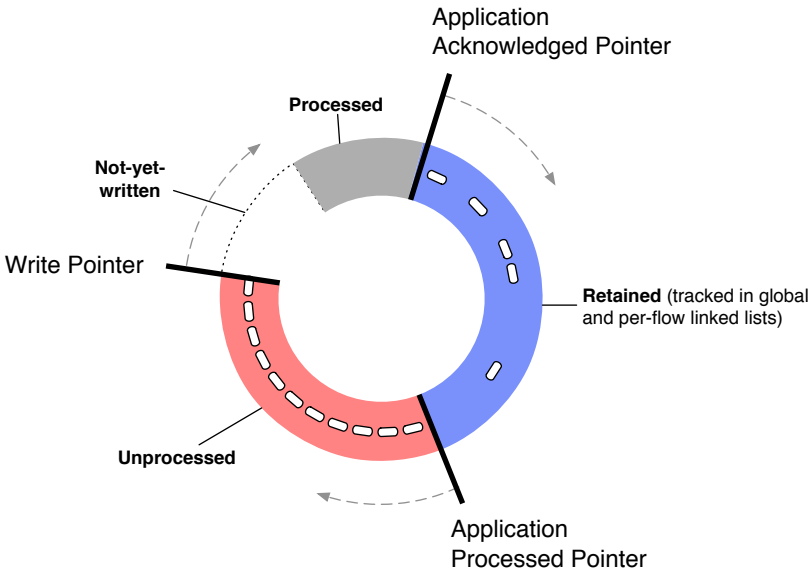


Figure 5.3: A revision of Figure 5.2 showing additionally the retained packets area ("Retained"). Note how the application now maintains two pointers: "Processed" and "Acknowledged". Note that the retained packet area is likely to be sparse—its lower bound is simply the oldest retained packet.

The per-packet processing required at this point is just a hash-table lookup and some book-keeping of packet queues and data counters (used for statistical purposes).

Once a packet arrives that represents the next segment to be delivered to the application, any retained packets are processed and delivered if appropriate. After delivery to a protocol analyser module, the protocol analyser may decide that it would like to buffer the packet, in which case it will be re-retained (or retained for the first time, if this is a newly arrived packet) and the protocol analyser will become the new owner. This scheme ensures that any need for data copying can be kept to a minimum throughout the lifetime of a connection.

If the main event loop determines that available space in the circular buffer is limited (due to the capture write pointer becoming close to the earliest retained packet), the owner of the retained packet (flow or protocol analyser) is given the opportunity to spill the packet out of the circular buffer and on to 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 to be stored. Since packet processing is incrementally data-driven, the amount of time taken to process each packet is bounded—protocol analysers are unable to block. As such, retained packets can be incrementally spilled to keep the ring buffer sufficiently empty. There is, however, a trade-off. Ideally, packets will be kept in the ring buffer for as long as possible without spilling, so that they can be processed, in order, without copying.

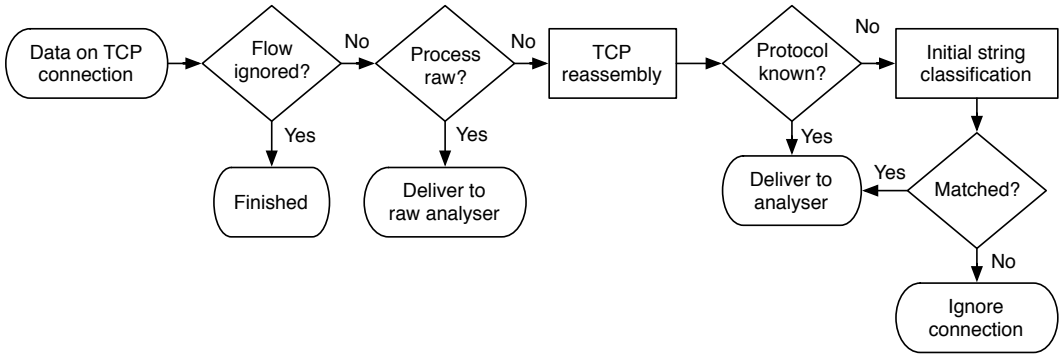


Figure 5.4: Data flow for TCP connections.

By considering the timing limits with Gigabit Ethernet, basic figures can be obtained for the duration of time a packet could remain in the buffer. For example, given a 1 Gbyte ring buffer and maximum 2 Gbps fill rate (full-duplex), the buffer will be filled in approximately 4 s. Therefore, a retained packet (with the current half-fill threshold), will have a minimum lifetime of 2 s, or proportionally longer for lower data rates.

5.3.3 Connection Processing

Figure 5.4 is a flow chart showing the broad stages of data processing for a TCP connection. If a flow has been marked as inactive, further processing on a packet is immediately abandoned. Assuming a flow is not ignored, raw packets may be passed to an attached protocol analyser. This is useful for analysis techniques that do not require full TCP reassembly to operate.

Following reassembly of out-of-order TCP segments, data is delivered in order to a protocol analyser. By default, a 'master' analyser is attached whose responsibility it is to identify the protocol by initial content inspection. Upon successful identification, the connection is handed off to the corresponding analyser. If identification fails, the connection is marked to be ignored.

The next section details the initial content inspection classification technique employed, and the following sections deal with the specifics of the abstract protocol analyser design.

5.3.4 Application Protocol Identification

Before a TCP connection has a protocol analyser attached, it is necessary to identify the correct protocol. This must be carried out for every TCP connection monitored by the system. If a connection cannot be identified within the first few packets or a limited number of bytes, it is abandoned. Once abandoned, a connection consumes minimal resources because it is no longer necessary to keep track of full TCP protocol state, although a record is still maintained in the connection hash table for

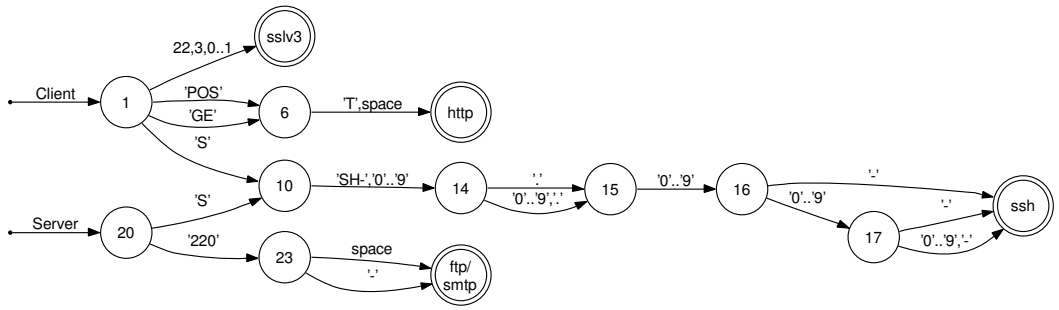


Figure 5.5: Graphical representation of finite state machine matcher for identifying application protocol from initial data. Each transition is labelled with its numeric byte or ASCII value within quotes. Ranges are given as nn..mm.

the lifetime of the connection to prevent repeated failed lookups and allocation of a state record for a partial[3] flow.

Protocol identification by payload inspection is generally based on scanning for signature patterns. Due to TCP segmentation, it is possible to miss a pattern when inspecting individual packets if the string of interest has been split across several packets. Intrusion detection systems take care to be robust in the face of such (potentially malicious) segmentation. Since only non-malicious use of known protocols is considered here, and IP fragmentation issues are ignored, it is sufficient to do simple initial-payload string matching after TCP reassembly.

Figure 5.5 depicts a simplified[4] finite state machine (FSM) that could be used to classify traffic. Double circles represent accepting states for SSLv3, HTTP, SSH and FTP/SMTP. The HTTP matcher, for example, matches the initial strings `GET␣` and `POST␣`. Note that the SSH string (such as `SSH-1.99-OpenSSH`) is valid from either client or server.

Incoming data from client-server or server-client TCP flow is passed through the corresponding FSM. If at any stage the data does not have a corresponding transition in the state machine, the TCP connection is immediately flagged as being ignored.

Figure 5.6 compares the performance of an FSM-based classification scheme with a sequential string match. The sequential matching (upper line) scales linearly with increase in the number of search strings. The FSM also scales approximately linearly in this range, but the correlation coefficient is so small that it is insignificant in comparison with the sequential matcher. As well as being able to express a wider range of patterns than an exact string match, a secondary advantage of the FSM approach is that it becomes trivial to scan across multiple TCP segments, since the state of the matching process is captured by the FSM.

Some protocols do not feature a 'magic' string that enables them to be un-

3. A flow for which the initial connection handshake has not been seen.
4. Nodes with a single in and out transition have been merged to save space. For example, strings of more than one character indicate omitted states.

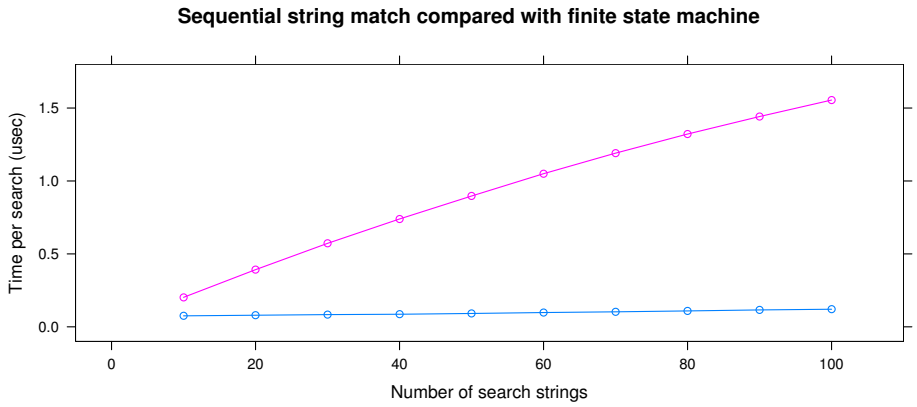


Figure 5.6: Performance comparison for sequential string matching (upper line) vs. a precompiled finite state machine (lower line).

ambiguously identified. In these cases, an approximate match against the expected fields in the header can be made. For example, the strings `220␣` or `220-` are not sufficient to differentiate FTP from SMTP. The full protocol analyser, once attached, will be able to perform stricter checks once it gains control. The initial string matching is intended to make a good first assessment of the application protocol for a connection, ensuring that the per-connection overheads are minimised if the majority of connections on the link do not need to be analysed.

Any TCP segments involved in the FSM-based application identification process are kept as retained packets, in order that they can be re-delivered to the specific protocol analyser once it has been assigned. It is important that the initial protocol identification phase is bounded in terms of the number of packets needed—both to limit the retained packet overheads and also to ensure prompt identification and processing of the content. If processing is delayed too much then associated auxiliary streams may commence before the control traffic that identifies them is parsed.

5.3.5 Basic Protocol Analyser Interface

A simple protocol analyser may operate at two different levels:

1. Raw un-reassembled packets
2. Packets delivered in order

The first case is straightforward and provides support for simple packet-based size or timing analysis.

The second case provides the full data from the connection. From the flow reassembly point of view, in-order data segments are delivered to the protocol analyser. If the protocol analyser determines that a block of data from the connection is not of interest, it may inform the reassembler to ignore any packets corresponding to a range of TCP sequence numbers. This allows, for example, an RPC-based

transport protocol which carries both control and data over the same TCP flow (e.g., the types of SRB flow that caused trouble with the Bro prototype from Section 5.2) to be monitored efficiently during both the control traffic and bulk-data phase. 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.

5.3.6 Threaded Protocol Analyser Interface

Implementing a protocol analyser that is driven by the receipt of data blocks of indeterminate size usually involves constructing an event-driven state machine (EDSM), where the events are 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.

In order to meet requirement R4, an alternative multi-threaded interface was designed, building on the in-order delivery service described in the previous section. The protocol analyser may be written as if it had access to two streams (one for each flow direction). Blocking read operations may be performed, and the analyser will be suspended until the requested amount of data has arrived. Therefore, imperative programming may be used to describe the control flow, whilst being based underneath on an efficient EDSM model.

Implementing such a system using normal OS-level threading primitives would incur considerable overheads if a thread were allocated per connection. Furthermore, pre-emptive multithreading would increase both development and runtime costs in the proper use of synchronisation/locking primitives. Therefore, an extremely lightweight user-level threading construct was employed. More details of the system used and an example of a threaded protocol analyser are given in Section 5.4.1.

5.4 Implementation

The system was implemented in C++. This gives the power and speed of plain C with the flexibility of an object-oriented language. The SRB analyser from Bro was written in C++, so it was a straightforward task to adapt it to work in the new monitoring system.

An Endace DAG monitoring card was used, and the software is linked with the Endace-supplied `libdag` in order to gain access to the DAG interface. `libpcap` was also used to provide a trace-file-based testing interface. Although some assumptions were made about the structure of the DAG packet capture header for reasons of efficiency, porting the system to operate with different hardware would

not be too difficult, since the actual interface to the capture hardware has been abstracted.

The timer manager was implemented using a custom calendar queue package, where future events are added to one of a circular set of buckets, indexed by the time until the timer is due to expire. These buckets can be thought of as days on a year-planner calendar. Each day may contain multiple events. In common timer implementations, strict ordering is required for event delivery. Here, per-second accuracy is sufficient since timers are mainly used for expiring TCP connections. This avoids the overheads of maintaining a time-ordered queue in each bucket.

The calendar queue buckets, as implemented, represent one second time slots. There can be hundreds of thousands of timers due to expire in a given second, so it is important to avoid processing all the timers at once because that could cause a buffer overrun on the DAG card. Instead, the number of timers in the next one-second bucket is examined every 20 ms (i.e., 50 times per second). Suppose the number of timers in the next one-second bucket is N, the current time is t and the deadline for the next bucket is T (a whole number of seconds). At each 20 ms interval the number of intervals available to process n timers is $i = (T - t)/0.02$, and so ideally $n = N/i$ timers are processed at a time.

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

A logging library, `rlog` [48], was used to provide a flexible yet efficient mechanism for logging. From the programmer's perspective the logging directives look similar to the familiar C `printf` function. Run-time configuration allows fine-grained selection of log messages, and, when disabled, a particular logging statement incurs minimal CPU instructions (corresponding to a load, test and branch).

5.4.1 ProtoThreads

As described above in Section 5.3.6, a lightweight user-level threading system is used to ease the implementation of the protocol analyser modules. It is based on the "ProtoThreads"[5] [39] library, which achieves thread switching by stack unwinding rather than traditional context switching.

The stack-unwinding process is conceptually based upon continuations—that is, the ability to save the state of an execution path as a continuation, proceed to execute other code and resume from the continuation later. In effect, a normal C subroutine is turned into a co-routine that yields when there is insufficient data to handle the next step of the protocol analyser.

By yielding, a C function is made to return (although this is hidden behind

5. "Proto" because they are small and not-quite real threads.

the ProtoThreads implementation). A drawback of this approach is that the state of local variables is lost between blocking reads on either TCP flow. However, C++ instance variables can be used to good effect to simplify the implementation. In addition, a side-effect of losing local variable state is that the memory required for the virtual context switch is minimised (compare this to a normal context-switching system where the entire processor state must be saved and restored).

```
1  int AnalyserClass::AnalyserMain()
2  {
3    EPABuffer buf;
4    EPA_BEGIN();
5
6    // Read function id
7    EPA_READ(buf,EPA_CLIENT,2);                          // YIELDS
8    func_id = *(uint16_t*)buf.data;
9    // Responder sends number of arguments as 16-bit int
10   EPA_READ(buf,EPA_CLIENT,2);                          // YIELDS
11   num_args = *(uint16_t*)buf.data;
12   for (arg_idx=0;arg_idx<num_args;arg_idx++) {
13     EPA_READ(buf,EPA_CLIENT,4);                        // YIELDS
14     arg_length = *(uint32_t*)buf.data;
15     // Read the argument, but only the first 200 bytes needed
16     EPA_READ_AND_SKIP(buf,EPA_CLIENT,
17                                arg_length,200);        // YIELDS
18     // ... process the argument
19   }
20
21   EPA_READ(buf,EPA_SERVER,4);                          // YIELDS
22   result_value = *(uint32_t*)data;
23
24   EPA_END();
25 }
```

Figure 5.7: ProtoThread example showing processing of an RPC-style argument list. Lines marked with **YIELDS** indicate points at which the function may yield.

Figure 5.7 shows a listing of an example ProtoThread-based analyser module. The code in question is intended to parse a series of length-prefixed arguments to an RPC call. A 16-bit function ID is sent, followed by the number of arguments (again 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.

In the listing in Figure 5.7, the `EPA_READ` and `EPA_READ_AND_SKIP` calls are C preprocessor macros that check if there is sufficient data available to satisfy the request. 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 `EPA_READ` statement.

Further details of the macro implementations are given in Appendix A.

5.4.2 Protocol Analyser Interface

Initial development of the SRB analyser was used to identify the basic requirements and programming interface for protocol analysis modules that interpret payload content. The non-SRB-specific parts were then extracted to form a general C++ base class (`ExpectingProtocolAnalyser`, "EPA") to be used by analysers of the form introduced earlier (Section 5.4.1).

When a new TCP connection completes the three-way handshake, the `ConnectionEstablished()` method is invoked on a new instance of the analyser class. Here the analyser is given the opportunity to inform the EPA base class how much data it is expecting to see on each side of the flow. So, for example, if the protocol dictates that the client transmit a 10 byte hello message to the server, an 'expectation' for 10 bytes would be issued on the corresponding flow. When that data arrives, the `DoDeliver(flow,packet)` method is invoked on the protocol analyser.

The EPA class provides macros that automatically manage the notification of expectations and combine them with the ProtoThreads library to enable the straight-line representation of the protocol structure. In particular, the following types and macros are available:

class `EPABuffer`

A simple wrapper for a data pointer and length value. Contains utility methods for extracting big-endian and little-endian integers. Used as the return value in the `EPA_` macros.

`EPA_READ(buffer,direction,size)`

Attempts to read *size* bytes from the *direction* flow, specified by one of the constants *EPA_CLIENT* or *EPA_SERVER*. The resulting data pointer is stored in the `EPABuffer` *buffer*. Each of the `EPA_*` macros may internally yield the ProtoThread, so it is unsafe to make use of local variables across them.

`EPA_READ_SKIP(buffer,direction,size,needed)`

Similar to the above, but now *needed* describes the actual number of bytes that the protocol analyser needs to see, whereas *size* represents the total number of bytes to pass over in the flow. This is used to efficiently skip large parts of a mixed control/data flow that are not of interest.

`EPA_READ_UCHAR(direction)` `EPA_READ_LE32/BE32(direction)`

These macros may be used in an expression to yield the next byte or little-endian or big-endian (respectively) 32-bit integer from the specified flow.

`EPA_READ_UNTIL(buffer,direction,stopchar,maxlen)`

Reads bytes until a specified character (*stopchar*) is found, up to *maxlen* bytes. This can be used for efficiently reading a line at a time. If possible the filled *buffer* will point directly into the packet remaining in the circular capture buffer or in the heap if the packet had been previously spilled. Otherwise, a

```
# Simple pattern for SSLv3
# First byte is content type id for handshake (22)
# Next bytes are protocol version (3.0 or 3.1)
ssl_v3 = 22 0x03 0x00..0x01;

# SSH protocol starts with version string like
#   "SSH-1.99-OpenSSH_3.8.1p1"
ssh = "SSH-" '0'..'9' {1,2} '.' '0'..'9' {1,3} '-';

client = (
    ssl_v3 @sslv3
  |
    ssh @ssh
  |
    ("GET " | "POST ") @http
);
```

Figure 5.8: Fragment of a Ragel input file for compilation to a C++ finite state machine implementation.

temporary buffer is allocated to store the incoming data up to *maxlen* bytes.

5.4.3 Application Protocol Identification

As described in Section 5.3.4, a finite state machine (FSM) is used to do initial identification of application protocol by examining initial content of the TCP flows. The FSM is described using a regular-expression like language and converted to very efficient native C++ code using an FSM compiler tool, Ragel [100]. Compile-time binding of the FSM is not considered to be a limitation here because the protocol analyser modules themselves must be compiled into the monitoring system. Addition of a new module involves creation of an appropriate branch in the FSM definition and corresonding code in a `switch` statement to instantiate the new protocol analyser when necessary. Figure 5.8 shows a fragment of the FSM input language showing patterns for SSLv3, SSH and HTTP.

Care must be taken when constructing the FSM to ensure that the pattern for one protocol is not a prefix of the pattern for another. This can be visually checked from the graphical representation (see Figure 5.5).

5.4.4 Event Reporting

Events of interest are delivered to a simple event sink class, the default implementation of which writes a line to a log file. Events may also be delivered across a Unix domain socket to a local process. This process, which could be implemented in a scripting language such as Perl, Python or Ruby, may act as a publish-subscribe system for passing the event on to a management system. Pre-processing may be also be implemented at this stage if the events need to be modified before forwarding.

The time latency between the real-time monitor dispatching an event and it being received by a local process is governed by the operating system scheduler.

On a multiprocessor system where the second processor can be dedicated to event processing tasks, this latency is expected to be less than 1 ms[6].

5.4.5 Testing

Unit tests exercise the normal and edge cases in the management of retained packets, TCP segment reassembly and delivery of data to protocol analysers. A basic-block coverage analysis was used to aid in identifying sections of untested code. The presence of these tests increases confidence in the accuracy of the monitor and also means that as the software is further developed any regressions can hopefully be identified.

A library of small trace files was created to support continuous verification of the tool's event-reporting output. These cover a range of application-level behaviour for the protocol analysers.

5.5 Evaluation

The preceding sections have described the requirements, design and implementation of the real-time application protocol analyser. The final part of this chapter evaluates some performance aspects of the tool under a synthetic test load and considers the scalability of the system. Section 6.5.1 in Chapter 6 completes the evaluation in the context of the implemented protocol analysers, from a real-time performance perspective and also in terms of detection accuracy. The analyser is shown to operate at speeds approaching 10 Gbits/s.

5.5.1 New Connection Rate

Once a TCP flow has been marked as ignored, the fast path involves only inspection of the packet header followed by a hash table lookup. 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.

To evaluate system load under extreme conditions for connection processing, TCP SYN packets were generated at a range of new-connection rates[7]. The rate was varied and the CPU load measured over a period of 90 s. The main purpose of this experiment was to gauge the limits of performance so as to obtain an approximate bound for the raw connection processing rate, irrespective of TCP reassembly or protocol analysis. The synthetic TCP SYN packets were sent across the monitoring link, such that each packet corresponded to a unique connection. A pseudo random number generator used to derive IP addresses and TCP port numbers was able to generate sufficiently unique traffic, as evidenced by expected active connection counts reported by the monitor.

6. Simple experiments showed the actual latency to be typically around 50 μs.
7. The monitor was positioned at point B on Figure 3.1 from page 32.

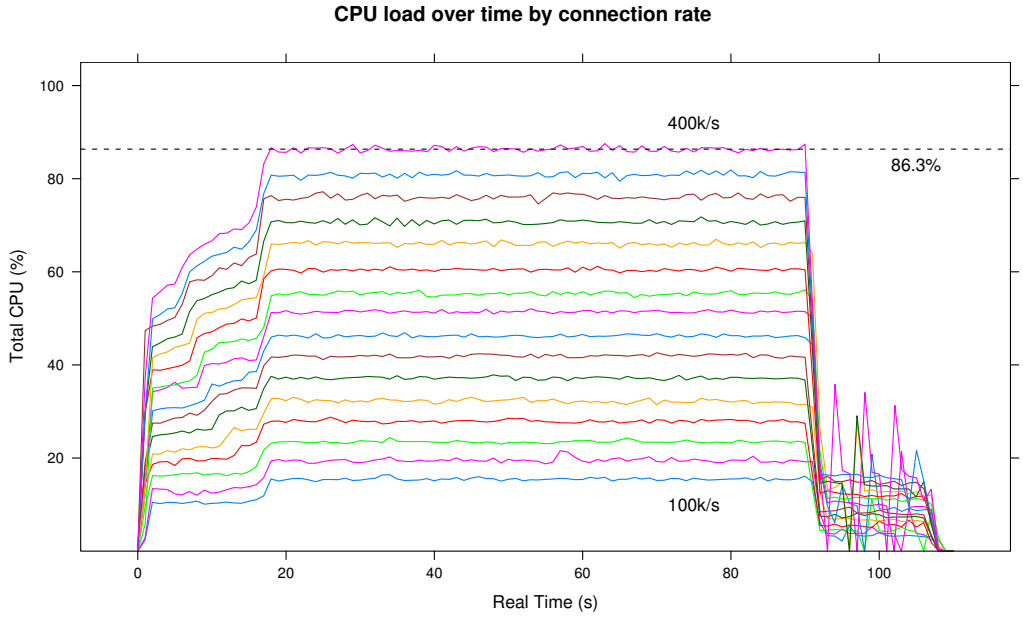


Figure 5.9: CPU load over time for different connection rates varied from 100,000–400,000 connections per second in steps of 20,000.

A single CPU on a dual Xeon 2.4 GHz was able to produce around 80,000 packets per second of synthetic traffic. Two CPUs on each of three machines brought the maximum packet rate that could be produced by the available hardware to around 480,000 packets per second. The theoretical limit on a full-duplex Gigabit Ethernet link is approximately 4 million packets per second[8]. Of course, a real network with that many new connections per second would be unusual, particularly when it is expected that most of the bandwidth is used by bulk data transfer applications. Even so, exceptional circumstances may occur if denial-of-service attacks are present on the network. What is important in these cases is that the monitor copes gracefully. This is dealt with in further detail in Section 5.5.5.

Figure 5.9 charts CPU usage[9] against elapsed time for connection rates ranging from 100,000 to 400,000 new connections per second. In each case the monitor was started cold.

After about 16 seconds the measurements show a sharp ramp-up before settling to an approximately constant value. This corresponds to a 15 second unestablished connection timer expiring and the start of regular expiry of connection records from the oldest connections[10]. At this point the number of active connec-

8. 1 Gbits/s $\times$ 2 $\div$ 8 bits/byte $\div$ 64 bytes/packet [minimum frame size, assuming packet bursting] = 3,906,250 packets/s
9. Both user and system time is counted, however system time is less than 0.5% throughout except during the initial ramp-up when memory allocations are being made. Here, system CPU usage peaks below 10%.
10. The discrepancy between 15 and 16 seconds is due to the coarse grained calendar queue timer implementation.

tion records stabilises at approximately $15 \times rate$. For a rate of 400,000 packets per second this corresponds to 6.4 million connection records. At this limit, CPU usage is around 86%. Noise after 90 seconds, when the load is switched off, is an artifact of timer processing when there are no regular incoming packets.

Each line shown on the graph represents a different connection rate, in steps of 20,000 new connections per second. It is clear that the relationship between rate and CPU usage is linear, which is largely governed by the hash table lookups. The hash table is not dynamically sized, in order to avoid processing stalls resulting from the rehashing that is needed during a size change. A larger hash table size would reduce collisions and therefore system load. Once collisions are minimised, the CPU load becomes more tightly linked to the analyser processing times.

5.5.2 Hash Table Sizing

It is up to the administrator to decide what hash table size is appropriate for the system configuration (e.g., memory availability) and the types of traffic of interest. The analyser can be configured to regularly log statistics relating to the connection hash table that may be used to inform the size configuration. For example, over each logging interval (typically seconds or minutes) the following values are given: the number of active connection table entries, the total number of lookup operations, the number of failed lookups and the average hash lookup depth encountered. The fill-factor can be computed from the hash table size and the active connection count. The number of lookup operations is roughly proportional to the packet rate. The average chain depth gives an indication of the collision rate, and any non-zero value is cause to consider increasing the hash table size, memory permitting. The table consumes four bytes per entry.

To give an idea of typical values, Figure 5.10 shows a series of graphs summarising various statistics from the new connection rate experiment. Figure 5.9, shown earlier, is a larger version of the lower-left chart ("Total CPU (User+System)"). The top left chart shows the average lookup depth over time, which stabilises at about 0.3 during the main part of the experiment. The hash table size here was about 23.5 million entries; the ratio against the steady-state connection count of 6.4 million yields a similar value (0.27). A discrepancy between these values would only be expected if the hash function were poor or the particular connection tuples encountered were to cause collisions[11].

5.5.3 IP Fragments

Assumption (3) on page 57 specified that the protocol analyser would not support the reassembly of IP fragments. Although this restriction should not be too difficult or costly to overcome given the flexibility of the retained packet scheme used by

11. This second condition might indicate a weakness of the hash function.

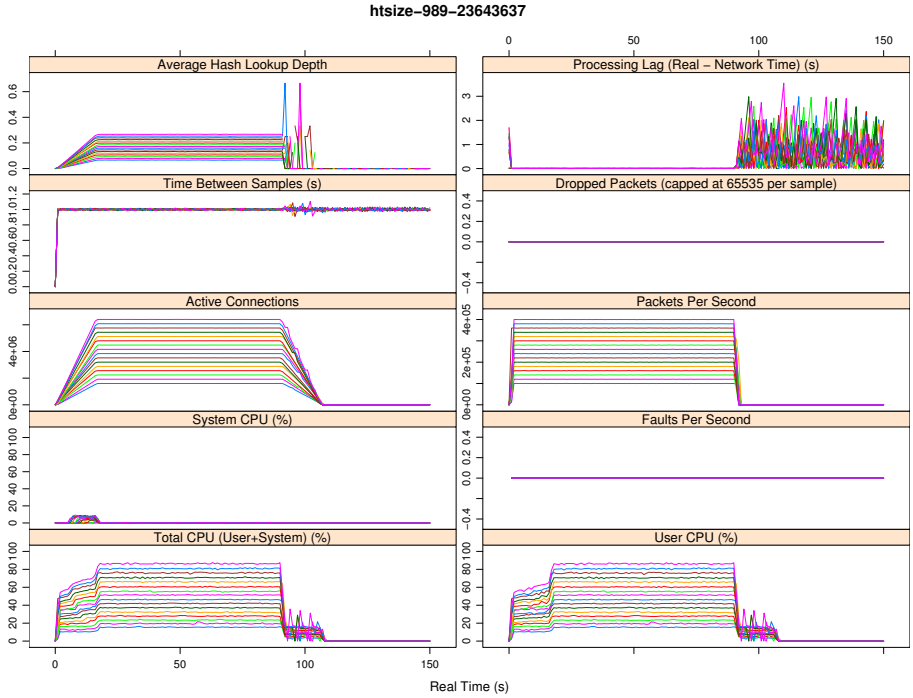


Figure 5.10: Connection rate experiment summary charts.

the analyser, it has not yet proved necessary. Emperical evidence to support this is that in a week-long trace from the DCS[12] edge router that contained 783 million IP packets, only 93 TCP packets were fragmented.

5.5.4 Scalability

Multiprocessing

The current system is inherently single-threaded. The data-driven main loop guarantees causality between the identification of related flows and their later appearance. The introduction of split-processing of the incoming packets—either via multiple DAG cards/buffers on a multi-CPU machine, or a cluster of machines with DAG cards monitoring the same network—would remove this guarantee. Parallel processing would mean that the start of a related flow might be processed before the appropriate protocol handler has been set up.

One solution would be to deal with *all* packets using the retained packet scheme, keeping each packet in a per-flow queue. Suppose that the workload is split amongst CPUs/DAG cards by a network-level hashing filter that delivers all packets from a particular flow to the same CPU. By keeping all initial packets on a flow in the circular buffer for around one or two seconds (see Section 5.3.2), sufficient time would be gained to send a message to the relevant monitor CPU. This would allow it to reinterpret the retained packets from a particular flow (assuming

12. Department of Computing Science

it has already started) before they have to be expired from the buffer.

The additional overheads of retaining packets should not be unmanageable, since only the first second or so of any given flow need be maintained—sufficient time for a message to be passed between monitoring systems.

Hardware Support

One of the design principles from Section 5.3 was to minimise memory accesses as far as possible. For example, packet content is only inspected by a protocol analyser when it is needed, and bulk data packets can be skipped entirely.

An aspect of IP and TCP that conflicts with this principle is the need for TCP checksum verification. IP header verification is not a significant problem since most of the header must be inspected anyway by the protocol analyser during the course of normal processing. The TCP checksum, however, is calculated across the entire payload and therefore can incur a non-trivial overhead.

The design of the monitoring system attempts to limit the overheads of TCP checksumming by avoiding or delaying the calculation until a packet has reached the point of delivery to a protocol analyser. Ignored flows do not need to be checksummed, and the checksum can be lazily verified before any operation that changes the state of the TCP flow in the analyser.

Hardware for TCP checksumming is commonly available in medium- to high-end Ethernet interfaces and could be implemented on an FPGA daughterboard on the DAG.

5.5.5 Points of Failure

Insufficient Memory

With 4 million new connections being created per second, an extreme version of the scenario covered in Section 5.5.1, a significant amount of memory would be needed to keep track of each of them. However, assuming that this level of traffic would be the result of a denial-of-service attack, it would not be necessary to actually maintain state for the connections. Instead, a limit of concurrent connections may be set by the operator in order to set an upper bound on the memory used by the monitor. Any new connections beyond this limit would be ignored. There is obviously a loss of information here, as the analyser is no longer able to interpret every session. However, even though this scenario is unlikely, graceful degradation and timely reporting of the situation to the operator means that benefit can still be gained.

If the monitor software were to allocate too much memory, the observed behaviour would be that the DAG driver software reports buffer overruns and there is a sudden drop in CPU usage by the monitor application. The first effect is due to the monitor application not servicing the packets in the capture buffer in a timely

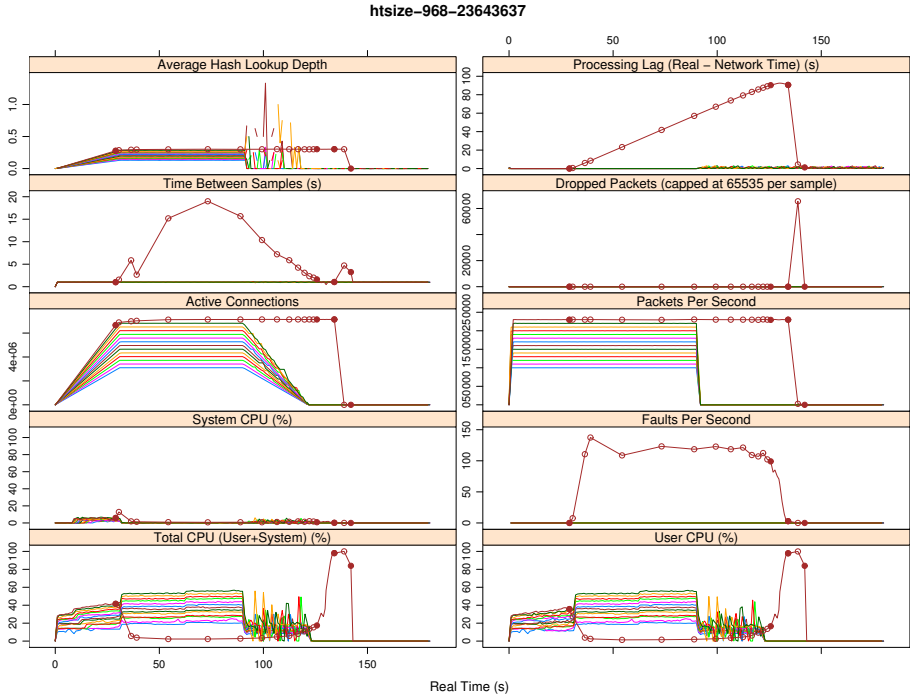


Figure 5.11: Connection rate experiment summary charts when swapping occurs.

fashion. This in turn is caused by the onset of virtual memory swapping by the operating system kernel on behalf of the monitor that is now using more memory than is physically available.

Figure 5.11 is an example of the operational statistics reported from the monitor in a low-memory situation. It is equivalent to Figure 5.10 except that a low-memory condition has been artificially induced. The process is blocked whilst virtual memory paging requests are handled, and as such the regular logging of the statistics used to generate the graphs becomes intermittent. Where the points used to draw the lines are greater than 1.5 s apart (the requested logging interval is 1 s here) the values are shown with circled points. A filled point indicates the beginning or end of a run of 'sparse' values. Note that most of the lines, corresponding to the lower connection rates, are unaffected. Only the lines representing the topmost rate (that causes memory requirements to exceed available physical memory) are abnormal.

The 'Faults Per Second' chart on the right hand side represents the cause of the unusual behaviour shown in each of the other charts. The 'Total CPU' chart in the lower left shows a sudden drop, which is perhaps unintuitive. The reason for this is that the process is no longer scheduled (until the paging request has completed) and therefore it appears to be using very little CPU time. Instead, the execution time becomes dependent on hard disk access times.

A further unintuitive result is shown in the 'Packets Per Second' chart. Here,

despite paging activity, the analyser appears to be keeping up with the incoming packets. In reality, the processing rate is calculated against the timestamp stored in each packet, whereas the X-axis of the charts is given in wall-clock time. Since the analyser is not able to keep up with real time, the DAG card gradually fills the ring buffer. The 'Processing Lag' (top right) chart represents the difference between real time and the time of the last packet seen by the analyser. The sudden drop at around 140 s indicates that the analyser has caught up with real time—but only because the DAG card has stopped capturing until the application buffer has been cleared ('Dropped Packets').

Lost Packets

The plain-text full payload protocol analysis described in this chapter is largely based on the parsing of full conversations between client and server. Any end-to-end packet losses between client and server will be detected and retransmitted by the end hosts' TCP, which means the retransmissions will be seen by the analyser. By using the DAG monitoring hardware, properly configured, packet capture losses at the passive monitor should be avoided. However, elimination of such losses cannot be completely guaranteed. These could be caused by reception errors at the Gigabit Ethernet protocol level that occur in the passive monitor but not at either endpoint of the tapped link. Alternatively, PCI bus contention may mean that the DAG card is unable to flush received packets to the circular capture buffer when necessary. Finally, speed limitations in the monitor software may mean that the contents of the capture buffer cannot be processed quickly enough, resulting in overruns. An acknowledgement (TCP ACK) for a segment that has not been seen is strong evidence for losses or asymmetric routing, and requires further investigation. The tool identifies such packets and logs them accordingly.

Since the system is intended to monitor network loads consisting of bulk data transfers, most of the packets seen by the monitor should be of the uninteresting variety. The design of the monitor is such that once a flow, or subset of a flow (identified by a range of TCP sequence numbers) has been marked as uninteresting, any packets falling within that range are completely ignored. The benefit of this is that any lost packets matching these criteria do not incur failure in the analyser.

If, however, a non-ignored packet is actually lost during the parsing of a known protocol, the analyser module will stall. This means that further packets on the same TCP stream will be retained, as happens in cases of genuine end-to-end packet loss. Once it becomes necessary for the earliest of these retained packets to be spilled, the protocol analyser can make a decision about how to resolve the situation. It may attempt to resynchronise against the data in the retained packets (therefore responding to the spill request by freeing the retained packet rather than actually spilling). Another option is for the analyser to abandon processing of the connection (and report this action accordingly to the operator).

All of the analysers currently spill packets rather than attempting resynchronisation. If retained/spilled packets exceed a preset per-analyser or global limit, the connection is abandoned. Such overflows should only occur in exceptional circumstances and, as described above, may be indicative of a configuration problem (e.g., losses in the Gigabit Ethernet optical tap) that should be investigated. During the course of the experimental runs, the behaviour has not been observed, other than during testing of the detection.

5.6 Summary

This chapter has covered the main design points of a real-time application-level protocol analysis system. Relevant implementation details have been explained, and the base system (ignoring specific protocol analyser implementations) has been shown to meet the required performance levels.

The next chapter examines the specific protocol analysers in detail and reports on their accuracy.

Chapter 6

Plain Text Analysis

Two different approaches are taken in the analysis of Grid bulk transfer traffic. Firstly, control traffic is precisely interpreted in order to report in real time the file transfer activity taking place. Secondly, heuristics are used to enable the classification of bulk transfer streams in the presence of encrypted control traffic.

Chapter 7 examines these heuristic techniques, while this chapter covers the content-specific control traffic analysis. This work has been carried out in the real time application protocol analysis system described in Chapter 5. The specifics of the technique are discussed first and then the capabilities of each of the implemented protocol analysers are covered in detail. Following this is an evaluation of the technique both in terms of meeting the event-reporting goals and satisfying the performance constraints. The scalability of the approach is considered and the suitability of the implemented system for application to emerging high-speed networking technologies is dealt with. Finally, the detection results from a week-long monitoring session are described.

6.1 Goals

The goals in performing control traffic analysis are:

G1. To identify the protocol/application in use without resorting to fixed TCP port number classification;

G2. To identify bulk-data TCP flows and attribute them to the corresponding control flow (and hence file transfer application); and

G3. To gather extra information about the bulk file transfers being performed.

For G3, file size information can be extracted as well as more detailed protocol-specific properties, such as the number of distinct file transfers being performed within a single control session and the version number of client and server software in use. Statistics aggregated from these may be used by ISPs to more clearly identify trends in network usage. As well as observing trends in intentional network usage, it may be possible, for example, to attribute anomalous traffic to broken or poorly implemented client or server software.

6.2 Costs and Limitations

In theory, monitoring of full protocol conversations—provided the conversations are not encrypted—permits reconstruction of the application-level behaviour, at least to a level that can provide useful data for network operators. Attaining this level of information by close inspection of packet contents would require a significant investment of time and expertise to cover all the traffic present on the network. This is because it is necessary to manually construct a protocol analyser module for each of the protocols in use. Development of an analyser module requires both an understanding of the protocol in question—at the network level—and also sufficient programmer skill to convert that protocol knowledge into the module code.

The costs involved can be reduced from two sides: firstly by limiting the range of applications under study to those which are expected to represent a significant network load; secondly by providing a framework for developing protocol analysers that reduces the amount of complex coding required. Chapter 4 introduced the set of bulk data transfer applications studied in this work, and Chapter 5 described in detail the framework for developing protocol analysers.

Analysis of control traffic by inspection of packet payloads is based on the assumption that packet payloads are available to the monitoring system. This carries with it an associated cost in terms of the additional system bandwidth required on top of the monitoring of transport-level packet headers only. For example, the packet capture hardware could be instructed to only copy headers into memory, which for Ethernet/IP/TCP would be 60–70 bytes. Compare this to full payload capture which might be expected to range from several hundred bytes up to 9000 bytes for jumbo Ethernet frames. In this work, the I/O bandwidth cost is weighed against the benefits of control traffic analysis (as demonstrated in this chapter).

Although packet content inspection raises concerns over security and privacy it is assumed that machine inspection of packet payloads for reasons of network management is not, ultimately, a privacy concern. The availability of a mechanism to exploit full content inspection for network management, along with evidence to support its effectiveness, could be used as an argument towards any policy changes necessary to support deployment of such a system.

6.3 Analysers

The following sections describe the specifics of each of the implemented protocol analysers, outlining those protocol features on which the analysers report and identifying any limitations present. In addition to these protocol-oriented attributes, any features of the protocol that dictate specific support in the protocol analysis framework are highlighted.

6.3.1 Event Structure

Each event described in the following sections contains the full or partial TCP flow tuple with which it is associated, as well as any parameters explicitly listed. Alongside this is an optional reference to another 'related' connection. This is typically set to the control flow that triggered the creation of the auxiliary connection record. For example, the textual representation of an event looks like this (split over several lines):

```
1139269632.05505991 BulkData:Created [192.168.1.23:33160 <-> *:0]
 {192.168.1.32:64985 -> 192.168.1.23:5021}
 id="5" source="BBFTP" ebytes_resp="522674"
```

The first field represents the event time (corresponding to the timestamp of the network packet that triggered it). An event name consists of a family name and event name, separated by a colon. Here the event family is BulkData and the name is Created. Within square brackets is the flow identifier this event directly represents (in this example, a pending connection to port 33160 of host 192.168.1.23). Within curly brackets is the related flow identifier (giving the corresponding control connection), which here represents a connection to a server on port 5021 of the same host. Remaining parameters are listed in straightforward `key=value` form.

6.3.2 Generic Bulk Data

All of the protocol-specific analysers take advantage of a shared component known as the 'BulkDataAnalyser'. This is a pseudo-protocol analyser module attached to flows that have been identified as bulk data but do not require any further interpretation of contents. The module counts bytes sent and received on the connection and generates an event when the analyser is created, when the connection is opened and when the connection is terminated.

Events

`BulkData:Created(source,expected_orig,expected_resp)`
: Issued as soon as the provisional connection is identified. The ⟨source⟩ parameter names the protocol from which the bulk data flow originates. For example, the textual sample given above in Section 6.3.1 is BBFTP.
If the protocol analyser was able to determine an estimate for the amount of data to be transferred over the connection, this is given by ⟨expected_orig⟩ and ⟨expected_resp⟩ (corresponding to bytes delivered from client to server and server to client, respectively).

`BulkData:Opened(source,expected_orig,expected_resp)`
: Issued once a connection has actually been established. It is possible for a Created event to occur without a subsequent Opened event, in which case a

Freed event will be issued after a timeout. This can happen if a bulk data connection has been identified by a protocol analyser but the transfer is aborted before the connection is established.

`BulkData:Closed(bytes_orig,bytes_resp,duration)`

`BulkData:Reset(bytes_orig,bytes_resp,duration)`

`BulkData:Freed(bytes_orig,bytes_resp,duration)`

Three possible events are given at connection termination: Closed, Reset or Freed. These correspond to the connection being gracefully closed by a TCP FIN handshake, being closed by a TCP RST segment, or being freed due to an inactivity timeout.

Resources

On top of the memory required by the TCP connection structure (see Section 6.5.2), the BulkDataAnalyser requires 88 bytes.

6.3.3 SRB Analyser

The SRB analyser that has been implemented is able to analyse the basic RPC system that underlies the SRB protocol. There are many functions corresponding to traditional file system APIs such as open, close, read, write, seek and stat. In addition to these, there are functions to 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.

The SRB protocol starts with a very brief TCP handshake on a fixed port (usually 5544) that negotiates another port number to which the client must reconnect in order to access the SRB server properly. The signature pattern of this broker connection is recognised in order to properly analyse the traffic on the second ephemeral connection. This second connection does have some regularity in the initial message exchanges that could be used to identify it as SRB traffic directly, irrespective of any previous broker connection. However, the broker connection is far simpler (containing just the string 'START SRB') and so reduces the complexity of the analysis that has to be applied to the start of every TCP flow.

The main connection begins with a software version and authentication handshake, after which the client takes control of a synchronous RPC exchange. The protocol analyser reports on the software version numbers in use and then identifies each RPC, with hooks at five points:

1. Once procedure identifier is known (i.e., which function is being called)
2. Before each argument is sent
3. After each argument has been sent
4. Once result code is known
5. After return value has been received

The basic SRB RPC protocol uses a set of straightforward 'read' and 'write' procedure calls to exchange bulk data. By adding logging hooks at the appropriate parts of the protocol exchange, events are raised in advance of the bulk data transfers. For example, a 'read' operation is reported at stage 3 once the number of bytes to be transferred is known. A 'write' operation is reported at 2 when the size of the argument is known.

Since bulk data and control traffic are mixed in this scenario (typically using 4 Mbyte transfer units), it is important for the analyser system to efficiently ignore the bulk data. This motivated the support for specifying a range of TCP sequence numbers for which incoming TCP segments should be dropped as soon as possible.

The SRB protocol has several different procedures used for reading and writing data—the particular one used depends on the version of the software in use and whether the transfer is client-client or server-server. The format of each of these procedures was identified by inspecting the native C SRB client and server source code as well as a plain-Java implementation of the protocol.

Beyond the simple RPC read/write mechanisms, newer versions of the SRB protocol are able to achieve much better network performance by separating the control and data connections. Bulk data can be streamed on the separate connection(s) rather than being periodically held up by repeated invocation of RPC calls. To implement this scheme, several SRB procedure calls identify the host name and port number to use for the additional connections. Depending on the particular run-time options, these may be from client to server, server to client, or third party server to client. The analyser module is able to parse these RPC calls and generate events to classify the auxiliary connection accordingly.

Events

In addition to BulkData events being generated with ⟨source⟩="SRB" for the identified bulk data connections, the following events are issued:

`Refer(port)`
: The result of the initial broker connection—specifies the port number to which the client will reconnect.

`Open(file_descriptor)`
: The client is opening a file. The ⟨file_descriptor⟩ will be used in subsequent `ReadChunk` or `WriteChunk` calls. This is the first indication that a control connection is to be used for combined control and bulk data transport.

`ReadChunk(file_descriptor,size)`

`WriteChunk(file_descriptor,size)`
: Read or write of ⟨size⟩ bytes of a previously opened file. This is reported before the transfer takes place.

`DataPut(host,port,size,num_threads)`

`DataGet(host,port,size,num_threads)`

Referral for secondary TCP stream(s) to be used for separate data transfer. BulkDataAnalysers will be created as necessary.

Resources

On top of the memory required by the TCP connection structure (see Section 6.5.2), the SRB analyser requires 220 bytes.

6.3.4 HTTP Analyser

Since HTTP itself is a very simple protocol, the analyser is similarly straightforward. Unfortunately, since HTTP is so common, it is difficult to obtain useful information from it without that information getting lost in the noise.

Basic parsing of the HTTP protocol is supported, including HTTP 1.1 connections with pipelined multiple transfers. For each transfer a small set of the response headers are logged as event parameters: `Content-Length`, `Content-Type` and `Server`.

The HTTP protocol is line-oriented, which means that parsing the stream for attributes of interest requires examining each byte of the request and response headers. If a `Content-Length` field is available then the size of the transfer is known in advance and the connection can be efficiently skipped until the next response. If there is no content length given, but 'chunked' transfer encoding[1] is in use, then each chunk can be efficiently skipped.

Byte-by-byte examination of packet payloads requires more memory bandwidth than the precise approach available when parsing the SRB protocol, for example, since message boundaries are determined by content rather than prefixed length fields. Protocols such as SMTP are more problematic in this regard, since the end of the 'bulk' payload (an email) is represented by a line containing only a period. Therefore, the data from each line must be considered.

Events

`GET(seq,content_type,content_length,server)`

Start of a GET request. ⟨seq⟩, a natural number, indicates which request of an HTTP 1.1 connection is being reported.

`Summary(count,orig_bytes,resp_bytes,duration,howclosed)`

Summary at the termination of each TCP connection. ⟨count⟩ indicates how many requests were issued. ⟨orig_bytes⟩ and ⟨resp_bytes⟩ are as given for BulkData events. ⟨duration⟩ is time in seconds and ⟨howclosed⟩ indicates the reason for connection termination ("FIN", "RST", or "timeout").

1. Chunked transfer encoding sends data in several length-prefixed chunks.

Resources

On top of the memory required by the TCP connection structure (see Section 6.5.2), the HTTP analyser requires 328 bytes. Of these, 130 bytes are used to keep track of Content-Type and Server header fields. These could be omitted for extra memory savings.

6.3.5 FTP Analyser

FTP, like HTTP, employs a line-oriented control protocol. The same underlying code for reassembling lines as used in the HTTP analyser is reused here. FTP is actually layered on top of the Telnet protocol, which interleaves some escape sequences with the application level messaging. A network intrusion detection system (NIDS) which needs to cope with the evasive techniques of intruders has to filter these out before scanning for FTP messages of interest, but since the classification of normal network usage is of primary concern here, run-time costs, development time and processing overheads can be reduced by ignoring such Telnet sequences.

Initial identification of the FTP protocol is accomplished by looking for an initial string from the server: '220' followed by a dash or space character (see Figure 5.5). Unfortunately, this simplistic approach contends with the hello message of the SMTP protocol. Therefore, an intermediate analyser is first attached which scans the initial messages exchanged on the connection. This allows the connection to be definitively identified as FTP before handing off control to the full FTP analyser. If this intermediate analyser encounters the handshake indicative of a GridFTP session, the GridFTP analyser is instantiated instead (see Chapter 7).

For plain FTP, the passive and active connection methods for data connection establishment are parsed and an appropriate BulkDataAnalyser is prepared and attached to a provisional connection record.

Events

`Port(port)`
: Notification that the control session has negotiated a port for connections back to the server.

`Host(ip,port)`
: Notification that the control session has negotiated a host and port for connections from the server to either the client or a third party.

`RETR(size,ip,port)`
: Start of a transfer from server to client. ⟨size⟩, if known from a preceding `SIZE` FTP command, is the size in bytes for the transfer. ⟨ip⟩ and ⟨port⟩ indicate one side of the connection to be involved. See also the generic BulkData:Created event.

`STOR(ip,port)`

Start of a transfer from client to server. ⟨size⟩, if known, is the size in bytes for the transfer. ⟨ip⟩ and ⟨port⟩ indicate one side of the connection to be involved. See also the generic BulkData:Created event.

Resources

On top of the memory required by the TCP connection structure (see Section 6.5.2), the FTP analyser requires 212 bytes.

6.3.6 BBFTP Analyser

The plain-text BBFTP analyser is designed to understand a subset of the client-server messages for BBFTP version 3. The source distribution does contain a document describing the protocol structure for each of the three protocol versions (1, 2 and 3) but unfortunately it is somewhat unclear and insufficiently detailed to give enough information for the development of a protocol analyser. As such, it was necessary to turn to the source code to better understand the protocol behaviour.

The client and server software are distributed as separate packages, each containing duplicated code that manages the network communication. Therefore, there is not one canonical reference point for the protocol. To further complicate matters, the source code is poorly structured which leads to difficulty in reverse-engineering the network-level behaviour.

Although the basic network exchanges are based on a regular message structure (consisting of type, length and content), the use of these messages does not follow a simple request-response sequence. In most cases, the messages of interest can be interpreted and other messages ignored by skipping the appropriate amount of data according to the length field in the message header. There is one message type in which the length field is reused for another purpose, thus requiring special-case handling.

Despite these disadvantages from the reverse-engineering perspective, the tool itself performs very effectively.

Similarly to the SRB analyser, the application-framed messages are decoded and the messages of interest are interpreted, resulting in updates to state stored in the analyser.

Events

In addition to BulkData attachments for data connections:

`Protocol(version)`

Report on the result of protocol negotiation.

`Retrieve(streams)`

A file retrieve is about to start with ⟨streams⟩ parallel flows.

`Store(size,streams)`

A file store is about to start with ⟨streams⟩ parallel flows and expected file size of ⟨size⟩ bytes.

Resources

On top of the memory required by the TCP connection structure (see Section 6.5.2), the BBFTP analyser requires 204 bytes.

6.3.7 iperf Analyser

Although iperf is not strictly a bulk data transfer application (in the sense that it does not transfer useful data), it is obviously still capable of inducing high network loads.

The network-level messaging used is very simple—as would be expected for a straightforward testing tool. The protocol structure is not documented other than in the source code, but the code is reasonably easy to understand and the relevant command-line options map fairly simply to network-level parameters.

The analyser interprets the initial client-server header that describes the parameters for the performance test being undertaken (number of parallel streams, duration, byte limit, etc.). This information is delivered in an event (see below) and then the control flow turns into a bulk data flow. Thereafter it is reported on by the BulkDataAnalyser module, alongside any auxiliary flows that have been identified.

Events

After the iperf header has been established, BulkData events will be reported for the same flow (with ⟨source⟩="iperf") and for auxiliary flows (⟨source⟩="iperf-server").

`Settings(flags,threads,port,winband,[duration|bytes])`

Report on fields from the iperf control structure. ⟨threads⟩ gives number of parallel streams in use. ⟨port⟩ is the auxiliary TCP port to be used and ⟨winband⟩ indicates the configured TCP window size. ⟨duration⟩ or ⟨bytes⟩ will be present depending on client run-time configuration.

Resources

On top of the memory required by the TCP connection structure (see Section 6.5.2), the Iperf analyser requires 168 bytes.

6.4 Sample Flow Statistics

As well as real-time traffic identification, the system can also be used in the generation of aggregate statistics over the analysed traffic. A key advantage of this type

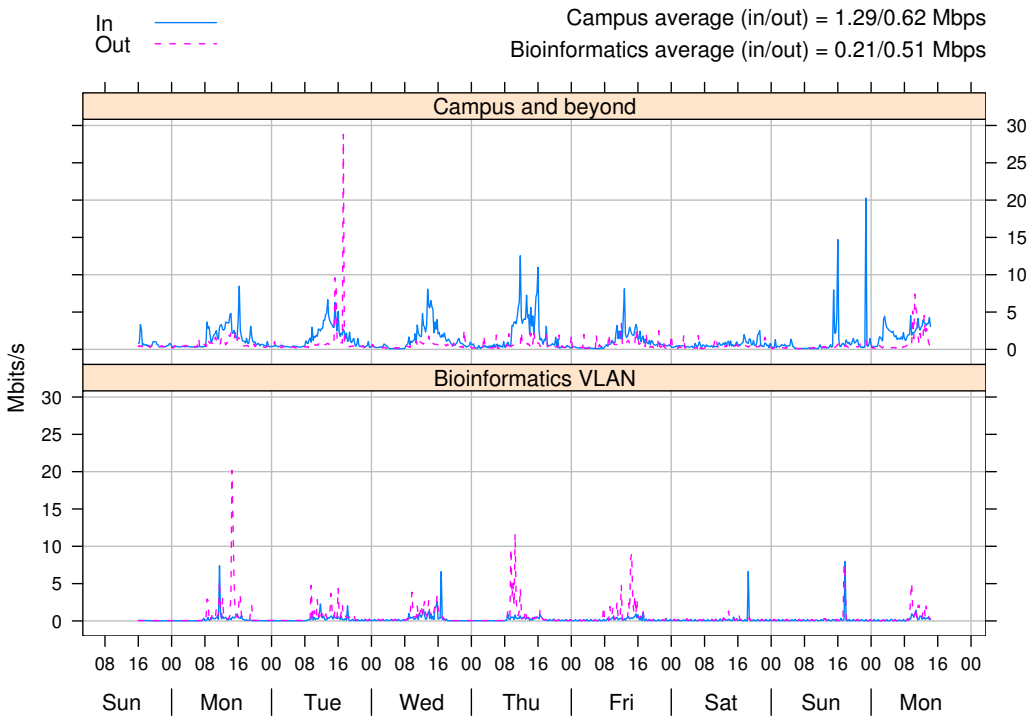


Figure 6.1: Network usage on DCS edge router between 20-11-2005 and 27-11-2005.

of usage over other traffic monitoring systems is that the protocol identification is precise, and can therefore be used to identify unusual activity on unexpected port numbers. In order to demonstrate the effectiveness of the system, statistics were extracted from a trace.

The GIGEMON hardware was configured to take a week-long trace of traffic passing the DCS router[2] between 20–27 November 2005 (Sunday to Sunday). The resulting trace of full packet payloads was 225 Gbytes. Figure 6.1 shows a traffic summary over this time period, including the split between traffic from DCS to the rest of the university campus and beyond to the internet, and between DCS and the Bioinformatics group . The average data rate for traffic for the rest of the campus and the further internet was 1.29 Mbits/s in and 0.62 Mbits/s out. Although analysers for SRB, BBFTP and iperf were enabled, no connections were expected or reported by these analysers because the applications are not in regular use within the department.

6.4.1 HTTP

Since the real time monitor is able to classify and interpret HTTP and FTP traffic by inspecting full packet payloads, this section presents a summary of the results produced by the tool. Corresponding with the diurnal variations seen in Figure 6.1, Figure 6.2 shows similar trends in the number of HTTP connections: significant

2. Point B from Figure 3.1 on page 32.

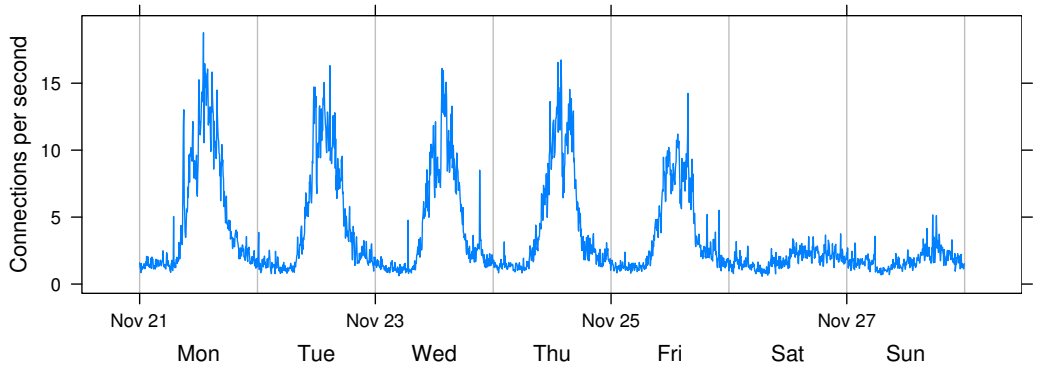


Figure 6.2: HTTP connection counts over one week.

peaks during the working days Monday to Friday with much lower activity at the weekend.

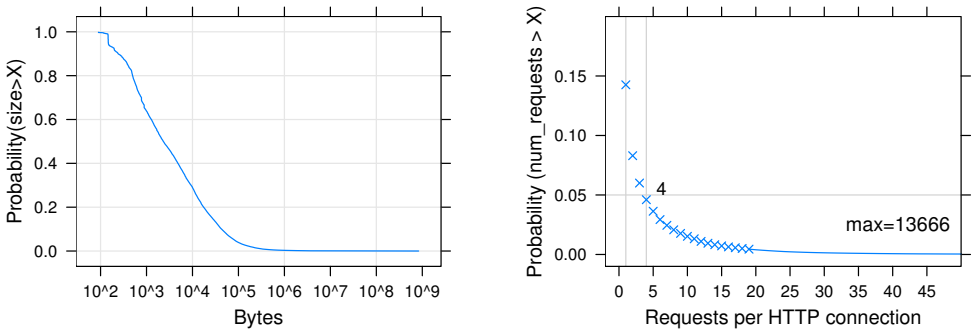
Of the HTTP connections identified, Table 6.1 sets out the server-side TCP port numbers encountered by the analyser. Unsurprisingly, HTTP's well-known port 80 is responsible for over 70% of the connections and 65% of the traffic volume. Sessions on port 8080 can be largely attributed to clients of the campus web proxy which listens on this port. Beyond these two, which together account for over 99% of the connections and data, a handful of other ports occur frequently enough to represent 0.01% or more of the connections. For example, traffic on port 443—which is intended for HTTPS traffic—was found to be non-encrypted HTTP traffic in 647 connections. Port 3689, used by the Apple iTunes LAN music sharing service, contributes 132 Mbytes. Beyond the port numbers shown in the table, another 45 were identified, together totalling only around 0.03% of the connections and only 11 Mbytes of data.

The HTTP protocol analysis performed by the monitor system keeps track of response sizes (i.e., the size of files fetched via the web) and the number of requests per connection (in the case of HTTP 1.1). Charts showing the cumulative distribution of response sizes and request counts are given in Figure 6.3. In the first of these, Figure 6.3(a), the X-axis is presented on a log scale. Fewer than 5% of the transfers were for amounts more than 100 Kbytes (10^5 bytes), and the largest transfer seen was around 1 Gbyte (10^9 bytes). The mean, median and standard deviation of this distribution were 41 Kbytes, 2.5 Kbytes and 1.5 Mbytes, respectively. In the interval 10^2 to 10^5 bytes the distribution appears to follow a logarithmic law, which tallies with other analyses of HTTP traffic.

For HTTP 1.1 sessions, where multiple requests may be issued serially on a single TCP connection, the analyser is able to track the request-response sequences. When large response bodies are sent the analyser can efficiently skip over any bulk data content by interpreting the presence of a `Content-Length` header field or the

Port	Count	%	Volume (Mbytes)	%
80	1,875,146	71.204	72,033	65.616
8080	749,244	28.451	36,933	33.644
2082	3,211	0.122	16	0.015
8052	2,006	0.076	13	0.012
8180	1,722	0.065	91	0.083
443	647	0.025	10	0.009
2869	473	0.018	1	0.002
3689	190	0.007	132	0.121
8069	15	0.001	347	0.316
8002	2	0.000	44	0.040
8035	1	0.000	126	0.115
8055	1	0.000	12	0.012
Others (45)	809	0.031	11	0.000

Table 6.1: Distribution of HTTP server port numbers. Ports contributing 0.01% or more by connections or volume are shown, and the table is sorted by connection count.



(a) Response sizes

(b) Requests

Figure 6.3: Cumulative distributions of HTTP responses.

HTTP 1.1 'chunked' transfer encoding. Figure 6.3(b) shows the cumulative distribution of the number of requests seen per HTTP connection. Note that the Y-axis scale here is limited to 0.2, and only 14% of the connections had more than one request; furthermore, only 5% had more than 4 requests. Beyond the X-scale of the graph are the relatively few connections with several hundred or more requests. The highest number of requests was seen in four connections with over 10,000 requests, which were attributable to a web-based newsreader performing regular updates over an extended period.

6.4.2 FTP

A similar treatment can be given for FTP connections. Table 6.2 summarises TCP port numbers seen for FTP control and data connections. The standard FTP control

Control Port	Count	%
21	10,658	77.6
8021	2,884	21.0
333	188	1.4

Data Port	Count	%	Volume (Mbytes)	%
20	2,107	15.3	305	2.4
Ephemeral	11,623	84.7	12,321	97.6

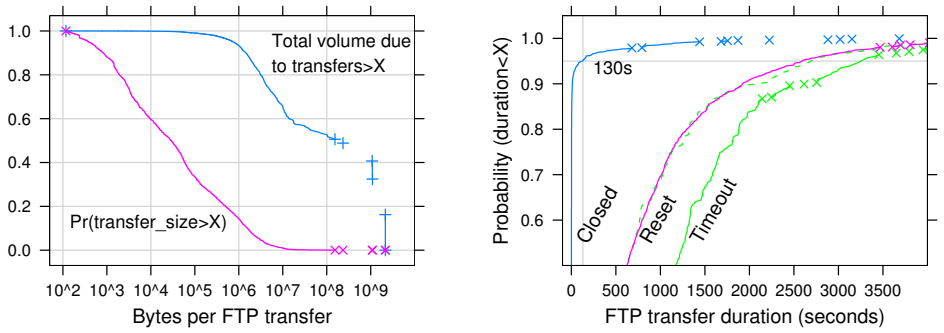
Table 6.2: Counts of FTP port numbers seen for control and data flows.

port, 21, accounts for over three quarters of the sessions seen, with just two other ports identified: 8021 and 333. Adding 8000 to a well-known port is common practice for running servers on non-privileged ports (those above 1023) so 8021 is not unusual. Port 333 is listed by IANA as 'Texar Security Port' and so in this instance appears to have been arbitrarily chosen by an FTP server administrator. Only 15% of the secondary FTP data connections are accounted for by the the well-known FTP data port, 20, and they are even less significant in terms of volume. Over 97% of the FTP data is carried on ephemeral ports, so this evidence strongly supports the need for dynamic analysis of bulk data traffic.

The distribution of file sizes for FTP is similar to that of HTTP, but with a significantly different range. Figure 6.4(a) shows two cumulative distributions of FTP transfer sizes: the lower line gives the probability that size of any single transfer is greater than x bytes, and the upper line gives the proportion of FTP data accounted for by transfers of size x bytes or more. The line is broken where data points are sparse. For example, there were only four transfers of more than 10^9 bytes, but these accounted for 40% of the total FTP data encountered.

Over the interval from 10^2 to 10^7 bytes, the transfer size distribution (lower line) appears to follow a logarithmic law, which extends to two orders of magnitude greater than that seen for the HTTP distribution in Figure 6.3(a). This corresponds to a limit of around 10 Mbyte for FTP compared with 100 Kbyte for HTTP.

Figure 6.4(b) shows the distribution of FTP transfer durations, grouped according to the reason for transfer termination. Note that the Y axis starts at 0.5. The far left line represents connections that were properly closed. The vast majority of these were under two minutes, with a few lasting as long as an hour. The remaining two categorisations are shown as 'Reset' and 'Timeout'. These correspond to connections that were closed due to a TCP RST segment being received, and those considered to be abandoned by the analyser due to an inactivity timeout, respectively. Clearly these abnormally terminated transfers tend to have lasted a lot longer than the successful transfers. This could be a result of users aborting a session that they



(a) Volume (b) Duration

Figure 6.4: (a) Cumulative distribution of FTP transfer sizes shown against traffic volume resulting from transfer sizes. (b) Cumulative distribution of FTP transfer durations, showing results for gracefully closed connections (Closed), abnormally closed (Reset), and those timed out due to inactivity (Timeout).

perceive to be taking too long.

The difference between the 'Reset' and 'Timeout' distributions arises from the analyser's active connection inactivity timer, which was set to 8 minutes during this run. Subtracting this interval from the 'Timeout' line yields the dashed line which runs alongside the solid 'Reset' trace. Since these distributions appear to be similar, it is likely that the split between termination reason arises from different software or hardware configurations, but is ultimately dictated by a similar process (e.g., user control).

In absolute terms, the 'Reset' and 'Timeout' flows together account for 59% of all the FTP flows encountered (50.3% and 8.7% respectively). This leaves just 41% of the observed FTP transfers completing successfully.

6.4.3 Elephants

The final results considered in this section are the events reported by a simple 'elephant' detector built into the real-time monitor. This detector operates on all TCP flows (whether allocated to a protocol analyser or not) and monitors data transfer rates over a fixed time window. An event is generated if the volume of data transferred during the window exceeds a pre-set threshold. For the data under consideration, the parameters were 5 seconds and 10 Mbits/s.

For traffic crossing the monitored link but not travelling beyond the campus network, the most frequently encountered services were, unsurprisingly, HTTP, SMB (Windows file sharing), and NFS. For internet-bound traffic, HTTP was again dominant in frequency. However, SSH sessions accounted for almost 4.5 Gbytes of traffic (compared to 3.7 Gbytes for HTTP), indicating the use of a secure file transfer protocol that tunnels data in an SSH session (e.g., `scp`, `sftp` or `rsync`).

Figure 6.5 shows the distribution of maximum transfer rates reported by the

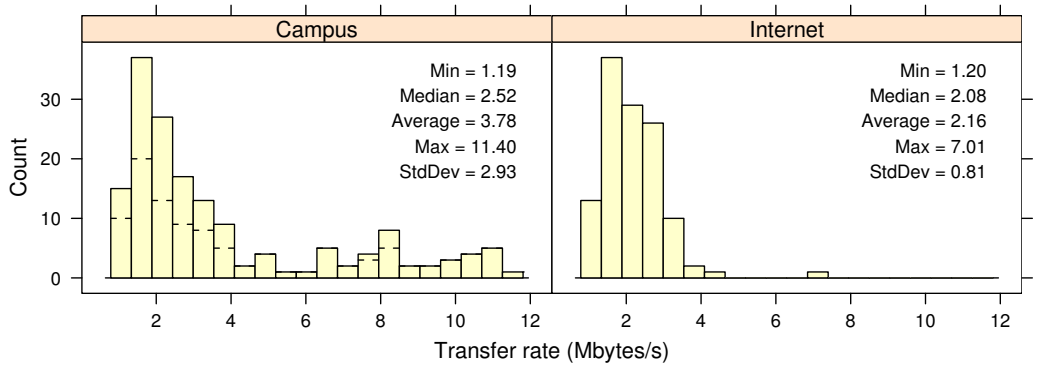


Figure 6.5: Maximum transfer rates for flows identified by the 'elephant' detector. Dashed lines on Campus plot show distribution after excluding port 8080 traffic.

detector for campus and internet traffic. The maximum rates seen, over a 5 second interval, were 11.4 Mbytes/s for campus traffic and 7.0 Mbytes/s for internet traffic. The concentration of transfers around the 2 Mbytes/s mark for both categories of traffic can in part be attributed to internet bandwidth limits. By excluding campus traffic from web proxies on port 8080, the dashed lines on the 'Campus' plot show more accurately the distribution of transfer rates on campus. The difference between the original profile and the dashed profile closely corresponds to the shape of the 'Internet' plot.

6.5 Evaluation

This section evaluates how the plain text analyser system meets the goals set out in Section 6.1, and revisits the requirements for the real-time monitor system from Section 5.1. The evaluation was incomplete in Chapter 5 because the specific instantiations of protocol analysers had not yet been introduced. Given these, Section 6.5.1 considers the performance of the system. Section 6.5.2 discusses the memory usage of the system with the implemented protocol analysers, and Section 6.5.4 examines the software development load for new analysers with a case study of the development process for one of the implemented analysers. Finally, Section 6.5.5 presents the results of an experiment to test the accuracy of the system.

6.5.1 Performance

Line Rate

The load on the monitoring system is heavily dependent on the mixture of traffic present on the link. For example, if the link is saturated with maximum-size packets from a few parallel TCP streams (e.g., from an iperf session or any of the file transfer protocols under study) then the CPU usage is less than 1%. The explanation for this

is that (a) large packets imply fewer packets per second; (b) few connections lead to fast hash table lookup; and (c) bulk data requires very little processing. On the other hand, a link saturated with HTTP connections places heavier load on the connection hash table and requires more CPU time in the analyser.

Sufficient hardware was not available to generate a synthetic load that would saturate the link but still be valid HTTP traffic. Instead, a subset of a trace from the DCS link was created that contained only traffic to or from TCP port 80. This trace was fed into the analyser (with packets being read from disk into a virtual DAG card ring buffer) and the user CPU time taken to process the trace was measured. System CPU time is not counted, as it corresponds to the system call overhead of reading from the trace file.

This sample HTTP-only trace was approximately 10 Gbytes and contained 12,370,868 packets and 204,324 distinct connections. Processing the file took 7.71 s on average over ten runs (with standard deviation 0.01 s), which translates to a rate of approximately 1.30 Gbytes/s or, equivalently, 10 Gbits/s. This is clearly far beyond the bandwidth of Gigabit Ethernet, and is approaching that available with 10 Gigabit Ethernet (bearing in mind that actual bandwidth is twice that because the link is full-duplex).

Therefore, for a typical mix of HTTP traffic, requirement R1 has been met.

Concurrent Control Connections

The previous section considered the performance limits when dealing with new connections—the allocation of new connection control records and their later disposal. Another aspect of performance evaluation is the number of concurrent protocol analysis sessions that can be supported. Here the term 'concurrent' is somewhat imprecise, since millions of dormant protocol analysers could be present even if they are not actively processing incoming packets. Therefore, of specific interest is the handling of concurrent control sessions that are active.

Under normal circumstances, the expectation is that most of the bandwidth of the link will be occupied by the bulk data being transferred by the applications of interest. As a result, the volume of control traffic should be relatively low. However, as with the new connection rate experiments, of primary interest is establishing how the system performs in extreme circumstances.

A trace-driven load of 4,000 concurrent SRB control sessions was generated by merging slightly time-shifted duplicates of a master trace consisting of several real concurrent SRB sessions. The separate TCP flow bulk data from these traces were eliminated and each duplicate control flow was assigned a unique IP address pair. Playback of the trace, ignoring the original timing information and sending packets as quickly as possible, was limited by the processing power of the load-generating machine. This resulted in 135 Mbits/s and 140,000 packets/sec of control traffic. During the handling of this traffic the monitoring system was parsing

Analyser	**Bulk Data**	**SRB**	**HTTP**	**FTP**	**BBFTP**	**iperf**
Size (bytes)	88	220	328	212	204	168

Table 6.3: Memory usage for plain text analysers.

the control-level protocol for each stream. The corresponding CPU usage peaked at 27%.

The SRB analyser 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 in general be left in the packet ring buffer and accessed directly from there once the complete RPC call has been parsed.

6.5.2 Memory Usage

Gigabit Ethernet is limited to around 4 million packets per second (see footnote 8 on page 70). Assuming that connections are active on average at least once per second, this quantity of packets per second can be converted into a bound for the number of concurrent connections: 4 million. If connections were active, on average, less frequently than once per second, the number of active connections expected would increase proportionately.

4 million connections without analysers attached (i.e., those which do not belong to one of the applications of interest) would require of the order of 800 Mbytes of memory, since each connection record consumes around 200 bytes. This could be reduced by using provisional 'mini' connection records that contain an order of magnitude less state, as is done in some OS implementations to cope with SYN flooding. These would be applicable to connections prior to becoming fully established via the three way handshake.

In general, memory usage by the monitor system occurs for only two other reasons beyond the base memory requirements of the TCP connection record structure (and associated hash table and timer entries). Firstly, memory may be allocated when a retained packet must be spilled from the circular capture buffer. Secondly, memory is allocated to maintain each protocol analyser's state, and a protocol analyser may in turn allocate memory in the course of its processing. Of the implemented analysers, only the SRB analyser makes heap allocations (to keep track of RPC arguments). The other analysers employ statically allocated buffers or are able to report on data as it arrives. Table 6.3 summarises the static memory requirements for each protocol analyser, which is typically just a few hundred bytes.

The decision to use static buffers (such as the `Content-Type` and `Server` fields from the HTTP analyser) is dependent on the size of the memory area re-

quired and the frequency of use and reuse. Clearly, infrequently used large buffers are good candidates for dynamic allocation, whereas frequently used small buffers are more suited to static allocation, especially when heap allocator overheads are taken into consideration (e.g., the GNU C library malloc implementation has a 4 byte overhead per chunk, and a minimum real allocation size of 16 bytes).

Retained packets are spilled when a threshold is reached for the amount of buffer space available to the DAG card for storing new packets. The threshold setting determines how much time is available to make a decision about what to do about the retained packet and then act upon that decision. In the simplest case there is a protocol analyser that is retaining a packet optimistically—for example, in case the content of a retained packet may be related to a packet/flow arriving shortly afterwards. By the time the packet is due to be spilled it may no longer be relevant. This provides a cheap timer mechanism for these situations.

If, on the other hand, the protocol analyser decides that it wants to hold on to the data represented by the retained packet, it is necessary to incur the cost of memory allocation and copying. The spilled packet is then the responsibility of the protocol analyser to free when possible. A more sophisticated technique for dealing with these situations—unimplemented as yet—would be to assign priorities to different types of packets that may need to be retained. These priorities could be evaluated across different protocol analysers, in order to effectively manage available memory in situations where free space is low. It may be appropriate to take different actions, for example: ignore new connections, drop existing connections, or prioritise existing connections and convert some to mini-records with less state.

Obviously each of these actions result in lost information. However, by preparing for such situations and acting appropriately the monitor can act in the most reliable way possible given the circumstances. By downgrading gracefully the operator has the opportunity to investigate what is happening and tune parameters appropriately to better deal with circumstances in the future.

6.5.3 Design Trade-offs

A consideration in the development of this kind of system is the conflict between achieving high performance and ensuring maintainability. High performance generally entails coding as close to the hardware as possible and limiting the levels of abstraction to avoid overheads of deeply nested function call-chains. Maintainability, on the other hand, dictates cleanly separated design and judicious use of abstraction to separate the layers of the system.

Both aspects have been dealt with by careful design, use of C++ language features, and allowing the compiler to do a good job of code optimization. For example, inline method implementations and limited use of virtual methods in the fast path allow the compiler to make compile-time decisions about which code may

```
1  iperf = (
2    (0x80 | 0x00) 0x00 0x00 (0x00 | 0x01) # High or low order bit set
3     0 0 0 1..255                         # 1−255 threads
4     0 0 any any                          # Any 16 bit port
5     0 0 any any                          # Any 16 bit bufferlen
6     any any any any                      # Any window size
7     ((0xff 0xf0..0xff any any)           # Up to about 3 hours
8        |
9      (0x00..0x7f any any any))           # Any unsigned byte count
10 );
```

Listing 6.1: iperf Ragel state machine definition

be used. Using inline methods in place of C preprocessor macros where possible improves code readability and considerably eases debugging.

6.5.4 Analyser Development Case Study

In order for the system to continue to be useful in the presence of new or revised bulk transfer applications, it is important to be able to easily implement and maintain protocol analyser modules (R4 from Section 5.1). As described in Chapter 5, analyser modules are represented by C++ classes. After initial string matching via the classification module, an analyser is attached to a TCP connection. In cases where this process is inexact, it may be necessary to add additional checks to the start of the protocol analyser to fully identify the connection (see, for example, Section 6.3.5).

This section describes the development of the iperf analyser module, in order to give an idea of the complexity involved. To begin with, the command-line tool was used in the normal way. This provided a basic understanding of the types of network connections involved, and from where the control channels are managed. As indicated in Section 6.3.7, a basic understanding of the protocol operation was gained from reading the source code. With this additional knowledge, simple `tcpdump` traces of the network traffic were taken between machines on the local network. These were manually inspected using a graphical traffic analysis tool (`ethereal`) and comparisons were made between the actual network behaviour and the predicted behaviour from the source code.

Having developed a good understanding of the network-level structure, the possible values for the initial bytes seen on the client-server connection were considered for the purposes of constructing a state machine (see Section 5.4.3) to match iperf traffic in general. Since iperf does not employ any obvious identification strings, it was necessary to identify the range of valid values for each field in the header structure and convert these to a sequence of byte value ranges suitable for use in the state machine.

Listing 6.1 shows the input to the state machine compiler corresponding to

iperf. Consider the final section, starting on line 7. This represents the 'amount' field of the iperf structure which is a signed 32-bit big-endian integer. If the value is negative it indicates a time period measured in 10 ms units; if the value is positive it represents the number of bytes to transfer in the test. The expression for the time period on line 7 (`0xff 0xf0..0xff any any`) was chosen to match a time period of up to around 3 hours[3]. Simultaneously, the state machine matches any byte count up to around 2 Gbytes (2^{31} bytes) (line 9), which is the maximum range for this field.

This pattern definition from Listing 6.1 can be examined to determine the likelihood of a false positive. The pattern contains several `any` components and broad byte ranges (such as `1..255`). Together, these account for just under 105 bits[4] from a 192 bit message. Therefore, the probability of a random match, assuming a uniform distribution of all possible messages, would be just $2^{105-192} = 2^{-87}$, or around 10^{-26}. Note that the pattern does not match the all-zeroes message (which could be expected to be relatively common) due to the minimum requirement of one thread (line 3).

Creation of this state machine description is a fairly straightforward transcription of the expected initial header values, taking care to make the matcher as specific as possible whilst at the same time being flexible enough to match the expected uses of the application in question. As explained earlier, any false positives at this stage could be eliminated by the more sophisticated control flow available in the proper protocol analyser, although this facility is not needed for iperf.

Once the initial classification was prepared, the analyser C++ class could be written. Apart from some common initialisation and book-keeping code, the iperf analyser consists of just one function, shown in Listing 6.2. The entire function is bracketed in calls to the macros `EPA_BEGIN` and `EPA_END` which hide the implementation details of the ProtoThread system.

An attempt is made to read the header block from the TCP flow using the `EPA_READ` macro. Upon return from this call, which may have blocked and only re-entered the analyser once enough data was available, the local variable `buf` represents the data available from the network. Since all the available header information is now present in memory, the Settings event is constructed via the `BEGIN_EVENT` macro. If a specific event has been disabled at run time, the code in the event block (lines 13–22) will not be executed. This mechanism ensures that the whole analyser system can be easily configured at run-time to operate as efficiently as possible, given the needs of the operator.

3. $-$(`0xff [0xf0..0xff] any any`) secs $= 1 +$ (`0x00 [0x00..0x0f] any any`) secs $\leq$ $1 +$ (`0x00 0x0f 0xff 0xff`) secs $=$ `0x00100000` secs $= 1048576$ secs $= 2.9$ hours

4. The total is not precisely 105 bits because the 1..255 range on line 3 contributes a fractional number of bits ($\log_2(255)$, approximately 7.99).

```
1  // Function to parse control flow from client to server.
2  PT_THREAD(IperfAnalyser::ParseMessage(struct pt *pt))
3  {
4      EPABuffer buf;        // Represents most recently read data
5      EPA_BEGIN(pt);        // ProtoThread setup
6
7      // Read iperf header structure (might block)
8      EPA_READ(buf, EPA_ORIG, sizeof(iperf_client_hdr));
9      iperf_client_hdr *hdr = (iperf_client_hdr*) buf.data;
10
11     // Deliver event
12     BEGIN_EVENT("Settings") {
13         event.AddInfo("flags","%08x",ntohl(hdr->flags));
14         event.AddInfo("threads","%d",ntohl(hdr->numThreads));
15         event.AddInfo("port","%d",ntohl(hdr->mPort));
16         event.AddInfo("winband","%d",ntohl(hdr->mWinBand));
17         int32_t t = ntohl(hdr->mAmount);
18         if (t<0) { // Negative means time limit
19             event.AddInfo("duration","%.2f",(-t)/100.0);
20         } else {    // Positive means byte limit
21             event.AddInfo("bytes","%d", t);
22         }
23     } END_EVENT;
24
25     // Check for connection back to server
26     int32_t flags = ntohl(hdr->flags);
27     if ((flags&IPERF_VERSION_1_7)!=0) {
28         // State machine (see listing 6.1) ensures numThreads has maximum of 255
29         int num_threads = ntohl(hdr->numThreads);
30
31         TCPConnectionKey key(conn->ConnectionKey()->addr_len,
32             conn->OrigAddr(), ntohl(hdr->mPort));
33         TCPConnection *newconn = new TCPConnection(
34             conn->Manager(), &key, NULL);
35         ProtocolAnalyser *bpa = new BulkDataAnalyser(
36             newconn, "iperf-server", num_threads);
37         newconn->SetProtocolAnalyser(bpa);
38         newconn->Install();
39     }
40
41     // Hand this connection over to BulkDataAnalyser
42     ProtocolAnalyser *pa = new BulkDataAnalyser(conn,"iperf");
43     conn->SetProtocolAnalyser(pa);
44     pa->ConnectionEstablished();
45
46     EPA_END(pt);
47 }
```

Listing 6.2: Source code to iperf analyser.

Within the event description block, information is extracted from the iperf header structure—including the time/volume `mAmount` field considered above in the context of the classification state machine.

If the header flags indicate that a connection is going to be made back to the client from the server, a BulkDataAnalyser is configured to recognise the forthcoming flow (lines 31–38). Finally, the flow at hand (the iperf control flow itself) is handed off to a separate BulkDataAnalyser in order that the bulk data to follow is properly reported.

The simple nature of the iperf protocol means that the analyser system can be also be fairly straightforward. However, the strength of the real-time monitor is that the framework handles the complexities of the network layer integration cleanly, and—perhaps more importantly—efficiently. The ProtoThread-based analyser design means that attention can be given to the structure of the analyser rather than, for example, managing buffering as would be necessary if data were delivered at the packet level. Although the flexibility of the system is not exploited fully by the iperf analyser, it is used effectively in the other analysers presented earlier.

6.5.5 Detection Accuracy Experiment

A blind experiment was carried out to ascertain the accuracy of the system in detecting flows corresponding to the implemented protocol analysers. A third party was asked to run a series of file transfers of their choosing using the applications under study and keep time-stamped logs. In parallel, the real-time monitor was run and the event logs stored. The monitor was positioned at the edge of the DCS network (point B on Figure 3.1).

Method

The experiment was carried out over a week-long period and consisted of 1,980 transfers. Most of the transfers were for files of size 10 Mbytes, purposefully small to avoid triggering campus-level anomalous traffic detectors (based on daily NetFlow reports). Although 10 Mbytes is small in comparison to the huge files that would be expected of a large Grid project, the precise nature of the detector means that it can recognise the traffic irrespective of transfer size.

Results

The application-level logs and the event output from the detector were correlated and it was found that the real-time monitor had 100% accuracy (no false negatives, no false positives) in detecting the flows that correspond to the supported analysers except in the unusual condition of long periods of control flow inactivity. If a control flow is inactive (i.e., no packets are sent or received) for longer than the inactive flow timeout then further activity will be ignored. This is because the late data

is considered to be an incomplete connection (the three-way handshake was not observed) and therefore not a candidate for full control-flow reassembly.

Problematic[5] inactive control flows occur in two particular situations: (1) when transfers are initiated in sequence over the same control connection, and any transfer but the last takes long enough to trigger a timeout; or (2) when some transient client or server problem causes an unexpectedly long delay. For example, the former may be experienced with serial FTP transfers. The latter situation was responsible for one missed event in the detection experiment, where an SRB server reported an error. The error unfortunately contained insufficient detail to explain the precise cause.

Flow Timeouts

The timeout period for inactive flows is a run-time configuration property of the monitor system, and applies to flows that have completed the three-way handshake. A separate, shorter timeout is applied to the initial stages of a connection, in order to effectively deal with port scan and denial of service traffic. Although it is not currently implemented, the results of this experiment suggest that provision for a dynamic timeout would be beneficial.

A dynamic timeout mechanism would treat flows differently depending on the protocol analyser that has been attached, if any. Uninteresting flows would receive a static timeout of several minutes, a value which gives a trade-off between tracking accurate flows for statistical purposes (e.g., flow duration and size distribution) and reducing resource usage in the monitor. A flow that is timed-out too soon will later reappear as a separate connection, therefore skewing such statistical information.

If a protocol analyser is attached, the flow timeout may be statically increased to an operator-supplied value (for example, one hour) to help ensure correct tracking of a complete control session. A further refinement would be to provide an API for the protocol analyser to inform the lower-level TCP connection tracking module of a desired timeout period. This timeout could be extended during the course of a file transfer (at which time the control flow would be expected to be inactive). The amount of extension could be another run-time configuration option, or a more accurate technique would be to use the presence of packets on a related bulk-data connection as a keep-alive for the control connection. Alternatively, a table of typical transfer sizes could be maintained, keyed by source-destination host. This could be used to inform the selection of timeout for future flows.

Letting the protocol analyser influence control flow timeout would be very effective for case (1) of serial transfers from a single control connection but would not aid case (2) where transient errors are involved. However, unless evidence from

5. A control flow can be inactive yet benign if, after a period of inactivity, no further transfers are performed

deployment were to suggest that such occurrences were non-negligible, it is unlikely that a more sophisticated system would be necessary.

Meeting Goals

This section considers how the goals set out in Section 6.1 have been met during the course of the experiment.

Content-based protocol identification (G1) was fully successful, since the control connections corresponding to each file transfer were correctly identified (including the differentiation between SMTP and FTP detailed in Section 6.3.5).

For each BulkData event reported by the analyser, the correct `source` attribute was given, indicating the control protocol responsible for its creation (G2). The bulk data flows corresponding to both single and parallel TCP streams were matched, and the total transfer reported by the `bytes_orig` and `bytes_resp` attributes tallied with the expected volume of data (obtained by multiplying the file transfer size, 10 Mbytes, by the number of transfers).

In addition to these basic requirements, the success of the system at identifying additional transfer information (G3) was evaluated. Generic support is present for providing the 'expected' number of bytes to be found on a flow, and is shown by the `ebytes_orig/resp` attributes of the BulkData:Opened event. It can only be populated if the control protocol communicates transfer size information before the establishment of the bulk data connections. For BBFTP this happens on every occasion, whereas for SRB it is dependent on run-time configuration. The specific commands invoked by an FTP client determine the availability of size information.

During the blind test, 13 Gbytes were correctly 'predicted' (i.e., the BulkData:Opened event correctly specified the approximate number of bytes on the flow) and 11 Gbytes were transferred with no prediction. In no cases was a prediction made which turned out to be false. The bulk data flows without predictions were due to FTP transfers in which the client did not issue `SIZE` command before the transfer or mixed control/data SRB sessions, in which the total transfer size is not known (however, the nature of the RPC mechanism means that each chunked transfer—typically 4 Mbytes—is identified in advance).

The above prediction results correspond to 53% of the data transferred being identified in advance. Of course, the precise mix of transfers which could or could not be reported on accurately depends on those chosen by the person running the file transfer applications. In some cases it may be possible to further augment a protocol analyser to extract additional information available. For example, some FTP servers will report the size of the file to be transferred within an unstructured message. Heuristics could be developed to parse these messages for likely sizing information to pass on to the appropriate BulkData events.

In any case, that the analysis system is able to report on any file transfer sizes in advance means that a management system is better informed to make reengi-

neering decisions.

Each of the implemented protocol analysers, described above, rely on the state-machine based classification system to associate them with the relevant TCP connections. The accuracy is therefore mainly dependent on the state machine representation for each protocol. In the case of the FTP analyser, however, the accuracy is dependent firstly on the state machine, but secondly on the module that differentiates between FTP and SMTP.

Beyond the evaluation undertaken with the blind experiment, the analyser has also been exhaustively tested with synthetic traffic (generated from scripted runs of file transfer applications) and this has failed to uncover any situations in which it does not work properly. These synthetic runs were designed to exercise a wide range of run-time configuration options available in the file transfer applications in order to test the protocol analysers' flexibility.

New versions or unusual uses of applications may not be supported by the protocol analysers. However, the log output of the real-time monitor may be used to identify the presence of such problems. If the occurrences are sufficiently frequent or it is suspected that the control sessions in question are responsible for significant bulk data flows then further manual inspection may be undertaken. The monitor system can be reconfigured to record captured packets to disk (with corresponding overheads) at the same time as the normal real-time analysis. The resulting trace files can then be studied offline using the appropriate tools to refine the protocol analyser.

Chapter 7

Encrypted Heuristic Analysis

The preceding chapter discussed the analysis of bulk-data-transfer control traffic by specific examination of packet payloads and reconstruction of full protocol conversations. As was demonstrated, this approach has the benefit of great accuracy but suffers from being inapplicable when payloads are encrypted.

The work presented in this chapter tackles particular cases of encrypted traffic by applying heuristic analysis techniques to the available network data.

7.1 Characterisation of the Problem

In Chapter 6 three key goals were specified: control protocol classification without using TCP port numbers, attribution of bulk flows to the controlling application, and gathering additional information about the transfers. The goals may be rephrased as questions to be answered when considering the use of encryption:

Q1. Can control flows still be classified according to the application in use?
Q2. Can data flows be matched with their corresponding control flow?
Q3. What additional information can be extracted?

It is clear that with the use of encryption it is no longer possible to achieve 100% accuracy in the identification of both control and data flows, and therefore it is necessary to consider the answers to these questions in terms of rates of false positives and false negatives. Two simple metrics can be used to make the evaluation: the number of bytes transmitted by a misclassified flow or simply the number of misclassified flows. Which of these is relevant ultimately depends on the actions taken by an operator in interpreting event output from the monitor. For example, if the policy is to completely block Grid-style bulk data traffic, the consequences of a false positive are much more severe than if a supposedly bulk flow is just routed over a separately provisioned link.

The remainder of this chapter investigates possible approaches and presents the results of applying a conceptually simple scheme to GridFTP file transfers.

7.2 Cryptography in Bulk Transfer Protocols

This section examines the use of cryptography (encryption, public key systems and message digests) in the applications previously introduced in Chapter 4. Each application is considered in order, starting with those using no encryption and moving on to the systems that represent a greater challenge for analysis.

The one application in the set that does not really qualify as a proper file transfer tool is `iperf`, and consequently it has no provision or need for cryptography.

There are two main variants of the SRB protocol worth considering. The first employs the Globus Security Infrastructure (GSI) to manage certificate-based authentication. However, the protocol is only extended to include the GSI credential exchange and subsequent message *safety*. That is, the main RPC phase of the protocol still exchanges messages in the clear but they are protected by a signature to avoid tampering. It is unclear why the full encryption capabilities of GSI have not been used. This SRB variant presents no problem for the plain-text analyser of Chapter 6[1].

The second alternative SRB protocol is the use of a custom-built security library known as 'SEA'[2]. It sits between the application layer and the socket (TCP) layer and manages message encryption after performing authentication. The core SRB protocol is unchanged; the RPC-style messages just become subject to encryption.

Even with these two potential control-protocol encryption methods, the data flows do not change: they are not encrypted nor authenticated beyond an initial 16-bit 'cookie', previously sent over the control connection. The SRB authors recommend compression and encryption of data sets on disk rather than at the file-transfer level.

BBFTP, like SRB, can employ GSI to manage authentication. Again, like SRB, it keeps the control protocol in the clear, changing only the initial credential exchange handshake. The plain-text monitor is able to interpret these connections and report on bulk-data flows and file transfer sizes. One configuration that does cause a problem is the tunnelling of BBFTP inside a secure shell (SSH) connection. The advantage of this from an administrative point of view is that if SSH service is already provided and managed, then authentication of BBFTP can be handled transparently (BBFTP itself does no authentication when operating in tunnelling mode, since it assumes that the user has been logged in appropriately by the SSH daemon). At the network level, a tunnelled BBFTP connection looks like any other SSH session because there are no clear-text-observable differences. Because the core control protocol remains unchanged across different transport mechanisms, so too

1. Although it is possible that the protocol will be altered in the future to use full encryption.
2. SDSC Encryption/Authentication

do the bulk data connections—i.e., they are not encrypted.

The RFC 2228 extensions to normal FTP (see Section 4.3) specify abstract provision for encryption and message integrity that may be implemented in any way mutually agreed by client and server. GridFTP is one such implementation. A GridFTP control session can be easily identified due to the use of GSI as the cryptography provider, and from server 'hello' strings representing known GridFTP server software. The individual FTP commands are encrypted and therefore not examinable, but the FTP protocol preserves the framing of the original commands, so it is possible to reconstruct a GridFTP session into request-response sequences.

As each data flow is established, there is an optional 'data channel authentication' process, which again follows the cryptography system negotiated by the controlling FTP session. Since GSI actually maps directly to TLS/SSL, the initial content of a data flow using authentication looks like any other TLS/SSL session.

As explained in Section 4.2, HTTPS uses TLS/SSL to transport a normal HTTP session. Therefore there are no specific clear-text attributes that enable it to be identified, which was also the problem with a tunnelled BBFTP session (although there the transport protocol was SSH rather than SSL).

GridFTP is the only application considered above, other than HTTPS, where encryption is certain. It has the advantage that it is clearly identifiable from control connection content, although data connections appear (according to initial string analysis) to be equivalent to other TLS/SSL flows.

7.3 Heuristic Analysis Approaches

The previous section outlined the ways in which cryptography is used within the applications under study. Some are completely clear-text, some just use standard public-key algorithms for authentication but not encryption, and others encrypt all or just the interesting parts of the control session. This has an influence on whether or not the control session can be identified according to the application in use. Recall the questions of Section 7.1, which query how well Grid-style bulk data can be identified in the presence of encryption. This section considers the available information and evaluates how it might be leveraged.

Before examining potential approaches in detail, it is useful to consider the 'baseline'—that is, the level of information that can be extracted without performing any complicated analysis at all. Assuming for the moment that it is possible to classify bulk flows as such according to some predefined threshold (for example, the elephant detector of Section 6.4.3), the problem then becomes the task of correctly identifying a corresponding control connection and hence the application. If a control flow has been identified (for example, as GridFTP), then its presence at the same time as a bulk flow between the same host pair could be treated as evidence for classification.

The drawbacks of this simplistic approach are as follows: bulk flows from applications other than the one for which a control flow has been identified would be mis-classified, and the identification must be delayed due to the use of a time window in the generic bulk data detector. It could be argued that the presence of any Grid-style communication between hosts is cause for all traffic between the hosts to be treated in the same way; whether this is reasonable depends on the management policy in place. Since such policy is out of the scope of this work, it is certainly worth pursuing a more accurate approach. Similarly, because timely reporting of bulk-transfer events is important for effective real-time traffic management, the time from the start of a connection to an event report should be minimised.

As mentioned in Chapter 4, some transfer protocols are capable of third-party transfers—where the client is not involved in the data transfer. Supporting the identification of these kinds of bulk flows would require the baseline matching rule to be broadened to say that *any* flow involving the server should be classed as Grid bulk traffic (not just those involving both client and server). This would certainly capture third-party transfers (provided their traffic passes the monitoring point), but would incur a substantial false positive rate for any other connections from the server. Furthermore, it creates the opportunity for abuse where an attacker could make a connection that looks like a known Grid bulk transfer protocol to a 'victim' server. The result would be that all bulk flows to/from the victim would be treated differently by the network management system. Again, the consequences would depend on the policy in place, which could be advantageous (and therefore an attacker might treat a host under his/her own control as the 'victim') or else constitute a denial-of-service attack. To reduce the scope of the investigation, third-party transfers are ignored.

Since the full application-level content is inaccessible due to encryption, the remaining properties that can be used to inform the identification of bulk data flows are message size, direction and timing, along with any information 'leaked' by the encryption system in use. As these features are representations of the original application-level messages, an obvious starting point is to attempt to reconstruct as much information as possible about the original messages. An understanding of the protocol structure might then be used to further abstract back to application-level phase changes. Clearly it will never be possible to extract IP addresses and port numbers without breaking the encryption, but by knowing the phase of the application (at least to some degree of confidence) the likelihood of correct bulk data classification could be improved.

The properties measurable at the network level are affected by several factors, which confound the reconstruction of application-level messages:

- Retransmissions/packet loss

- Segmentation resulting from path MTU
- TCP stack configuration parameters
- Communication library framing and messages
- Encryption algorithm padding

Retransmissions and segmentation could be filtered out with TCP reassembly, but the resulting timing fluctuations would remain. The extent to which such fluctuations could be ignored depends on their magnitude relative to the higher-level timing. For example, the timing of messages from an application that only transmits or receives every 10 seconds or more would mean that retransmission timing errors would be negligible. Unfortunately, such intermittent behaviour in a file transfer protocol would be unusual and therefore timing errors introduced by retransmissions must still be considered. Analysis of specific losses could yield a 'cleaned-up' timing profile, by interpolating times for missed packets and adjusting responses according to the delay incurred.

In order to infer application-level message sizes from the observed network traffic, a set of packets must be combined into a logical message. A simple scheme for doing so is to treat a sequence of packets in one direction as a unit, bounded by the receipt of a message travelling in the other direction or the start of the connection. A refinement is to treat gaps in packet receipt, according to some time threshold, as a message separator. This is important where a message is the trigger for a bulk transfer and the control session is idle until the end of a transfer.

An alternative technique is applicable when the encryption mechanism uses a record format with identifiable boundaries, as is the case with TLS/SSL, GridFTP, and SSHv1 (but not SSHv2). The application-level message sizes can then be directly extracted, subject to errors incurred by padding in the encryption algorithm (using an 8, 16 or 32 byte block size, for example).

The discussion above has indicated the type of raw data available for investigation of an encrypted control channel. Any analysis of such data must be achievable in real-time, and also be applicable to the few message exchanges expected in a bulk file transfer protocol—these usually correspond to an authentication exchange, some setting of transfer parameters, and a request for a particular file referenced by some identifier, such as a file name. In particular, any analysis technique must be able to produce results incrementally, without having to wait until the completion of a flow (since in this case clearly it would be too late to raise any events).

The next section investigates the properties of GridFTP file transfers by visualising the control channel timing and size information in a series of charts. These then support reasoning about the potential for time/size-based analysis.

7.4 GridFTP Timing Analysis

A series of experiments were undertaken to investigate properties of GridFTP control sessions. They were carried out using scripted runs of the command line tool `globus-url-copy` running on a machine at DCS and connecting to `bananarama` at Lancaster.

The intent was to observe the range of message exchanges between client and server, considering both the encrypted messages found at the network level and the plain text messages being exchanged at the application level.

By setting appropriate environment variables before running the client program, a log file was generated consisting of the plain text and encrypted messages passing through the GSSAPI encryption library. Messages from this log file were correlated with those extracted from the network-level trace, from which the message timing is obtained.

For example, when the client sends the plain FTP command `SITE HELP`, an encrypted message of the following form is produced:

```
ENC FwMAACBOTRjzEJphk2l+5UVgpG/VPtffkN+KdNvWUlliUwGyig==
```

This is an `ENC` command containing a base64-encoded encrypted payload.

7.4.1 Method

To get an idea of the variations in the protocol and the corresponding effect on the observable network behaviour, the `globus-url-copy` program was run with each of the following range of options:

`-a`	ASCII transfer mode
`-dcpriv`	Data channel privacy (encryption on data channel)
`-dcsafe`	Data channel safety (signed hash authentication of data)
`-nodcau`	No data channel authentication
`-fast`	Use extended FTP transfer modes
`-p 1`	Parallel data mode, one channel
`-p 4`	Parallel data mode, four channels
`-r`	Recursive directory copy

The program was also run without any special options, and in this case four files with *names* of different lengths were transferred (10, 30, 50 and 80 characters). This is relevant because the full file name/path is sent across the control connection, and hence the size of the encrypted messages varies.

For the parallel modes, three files of different sizes were transferred: 1,000 bytes, 1,000,000 bytes, and 10,000,000 bytes. The small sizes of these files (compared to the types of file that might be expected to be transferred over GridFTP) meant that running the tests was a low-cost exercise, in terms of time and disk storage.

Furthermore, these file sizes have the property that representing them in decimal each requires a different number of characters—again something expected to be seen at the encrypted message level.

The recursive mode transfers were of a directory containing 10 files of 1,000,000 bytes each.

The set of different configurations was run 10 times per hour over the course of one week-day (a Wednesday). The 10 runs were carried out in sequence, with each taking about two minutes to complete each combination within the set.

7.4.2 Interpreting Results

Inspection indicates that the `-a` mode has no effect on the control channel, and that `-fast` is equivalent to `-p 1`.

In order to visualise the message sequences observed, a set of graphs were produced to show the encrypted and plain text request response sequences. Two different types of graph were considered: one that ignores real timing and simply shows the message sequence; the other shows real time. Examples of these are shown in Figures 7.1 and 7.2, which correspond to the default options and transfer of a file with 10 characters in the path name.

The lines and points in the graphs represent an FTP-level message. The y-position indicates the message length; positive indicates client-server, negative is server-client. In Figure 7.1, green triangles indicate the actual plain text message lengths and red circles indicate the message length obtained from inspecting the encrypted payload. In each case the red line has greater-or-equal magnitude to the green. The maximum error between encrypted and plain-text messages is indicated at the top left by 'Max delta'. This delta is due to padding in the encryption algorithm. For the algorithm in use (3DES-EDE), it should be less than 16.

Above each client-server point (positive) is an annotation showing the first word of the corresponding FTP command[3]. The immediately following server-client messages may be considered to be the server response to the command in question. The server sends each line of its response as a separate encrypted message. For example, the response to `SITE HELP` (the third client message in Figure 7.1) is:

```
214-The following SITE commands are recognized (* =>'s unimplemented).
   UMASK           GPASS           ALIAS           BUFSIZE
   IDLE            NEWER           CDPATH          PSIZE
   CHMOD           MINFO           GROUPS          FAULT
   HELP            INDEX           CHECKMETHOD
   GROUP           EXEC            CHECKSUM
214 Direct comments to ftp-bugs@tim.
```

The corresponding messages are the negative points between `SITE` and `FEAT`. The

3. For example, `SIZE /some/file` shows as simply `SIZE`.

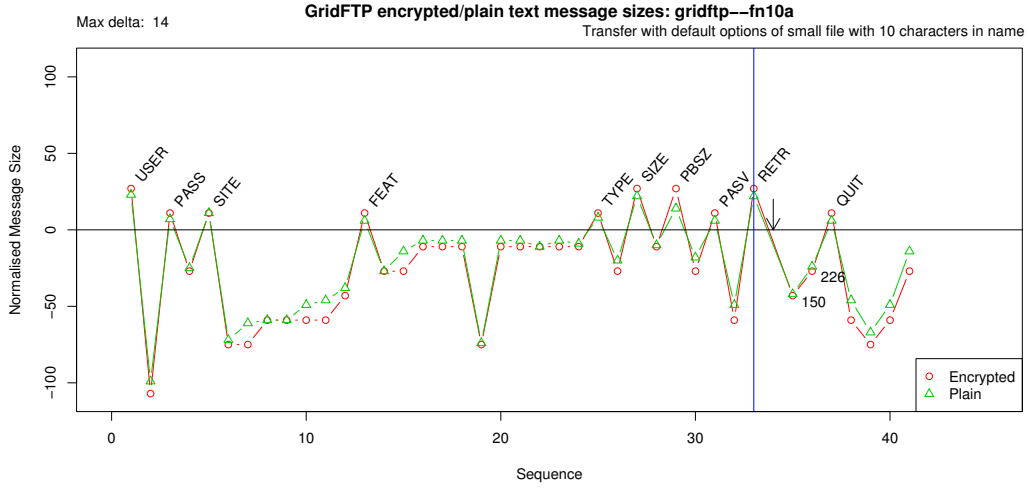


Figure 7.1: Sequence of messages exchanged by a GridFTP client and server. Values on y-axis indicate size of message, showing both real message size and the quantised size inferred from the encrypted payload. Positive values indicate message from client to server. The name of the file being transferred was 10 characters in length.

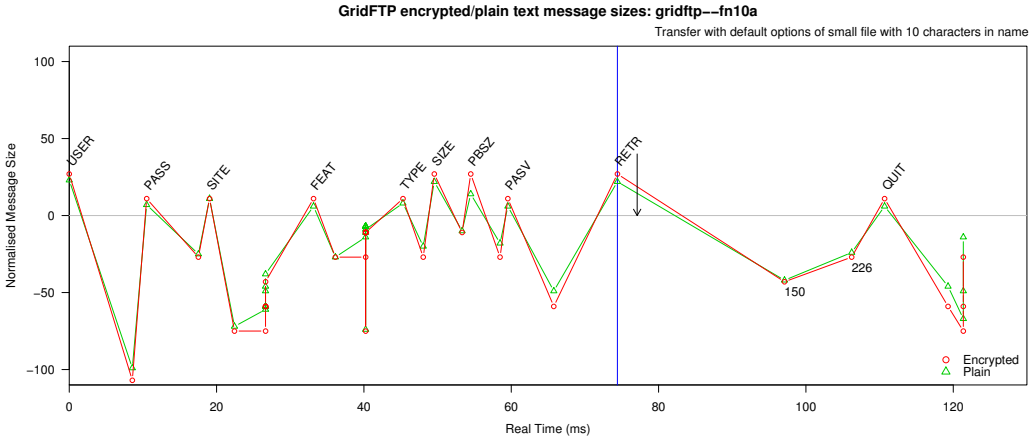


Figure 7.2: Sequence of messages exchanged by a GridFTP client and server. x-axis shows real time. Compare with Figure 7.1 where x-axis shows sequence number.

chart also shows that the response to the FEAT command has a long line in the middle (sequence number 19), as evidenced by line 6 in following extract from the transcript:

```
211-Extensions supported:
 REST STREAM
 ESTO
 ERET
 MDTM
 MLST Type*;Size*;Modify*;Perm*;Charset*;UNIX.mode*;UNIX.slink*;Unique*;
 SIZE
 CKSM
 PARALLEL
 DCAU
211 END
```

Figure 7.2 shows the same set of messages with a real-time x-axis rather than integer sequence numbers. A small arrow is shown just after the RETR command, pointing towards the x-axis. This arrow indicates the initial SYN packet of an auxiliary data flow. The direction of the arrow indicates whether the connection was client-server (up) or server-client (down). The vertical blue line highlights the RETR command.

Note that the messages at time 40 ms are overlaid—this is because the entire encrypted response to the FEAT arrived in the same packet, hence each message received the same time stamp. A similar phenomenon occurs for the SITE command.

The timings shown in this chart are slightly skewed since the packet trace was taken on the client machine. Therefore, the delay between server response and next client message is generally very small, whereas the time between client message and server response is governed by the RTT. If the trace were taken somewhere closer to the middle of the network, these measurements would be more balanced.

The chart appears to indicate that the response to the PASV command and subsequent issue of the RETR command takes longer than the preceding TYPE, SIZE, and PBSZ commands (at around time $x = 70$ ms). The response to the PASV command is:

```
227 Entering Passive Mode (192,168,1,23,136,63)
```

Note that a server-side listening socket has been created, and the IP address and port number have been communicated in this message to the client. It is tempting to attribute the extra time to the kernel and network stack opening the socket on the server. Indeed, a similar profile has been observed when the client allocates the socket and notifies the server via the PORT command. However, the chart shown corresponds to a single file transfer, and is not representative of the complete set of traces. The aggregate properties of the traces will be dealt with in the next section.

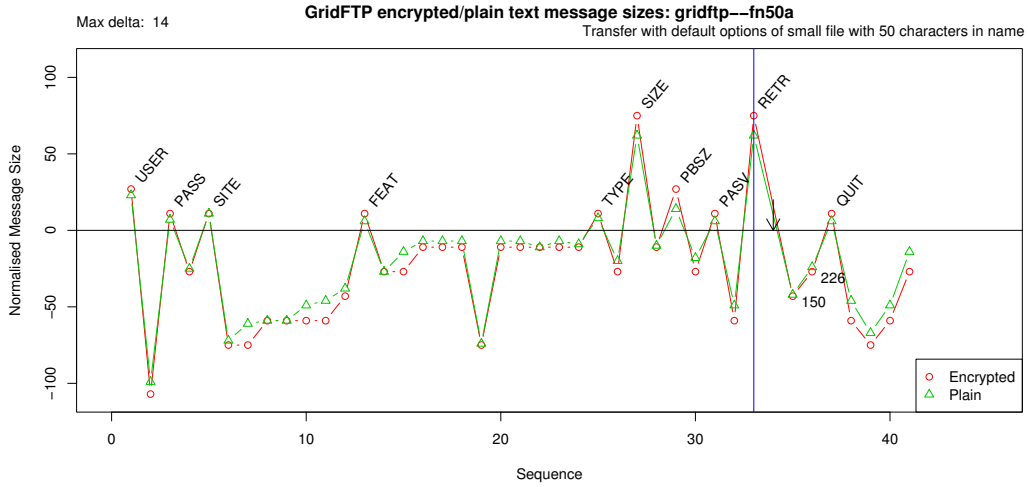


Figure 7.3: As Figure 7.1 but for a file name of 50 characters. Note that the `SIZE` and `RETR` messages have increased accordingly, but all others are identical to the previous figure.

Recall from Section 7.4.1 that the file transfers were carried out both with different file path lengths and file content sizes. Figure 7.3 shows the sequence plot for a file of path length 50 characters. It is identical to Figure 7.1 except that the message length for the `SIZE` and `RETR` commands is larger. The sequence plots for the larger file transfers (larger in terms of number of bytes) are also different, but more subtly so.

Figure 7.4 shows close-ups of extracts from the sequence charts for the different file sizes. The plain text messages take the following form, where > and < represent messages from client and server, respectively:

```
> SIZE fileA          > SIZE fileB          > SIZE fileC
< 213 1000            < 213 1000000         < 213 10000000
```

Since the number of characters required to represent these file lengths increases, there is a corresponding increase in message size. The figure shows, however, that the observed message size—as can be inferred by looking at the encrypted payload—is quantised such that it does not change between the one million and 10 million byte files. Although it is certainly present, the one character difference in the corresponding plain text messages is hard to discriminate from the figure.

7.4.3 Aggregate Timing

The previous section dealt with the initial interpretation of the GridFTP messages in terms of the sizes and timings of messages and introduced the potential for timing information to be used to better identify the phase of the GridFTP session. This section gives a more rigourous evaluation of the additional delay observed around the `PASV` and `PORT` commands.

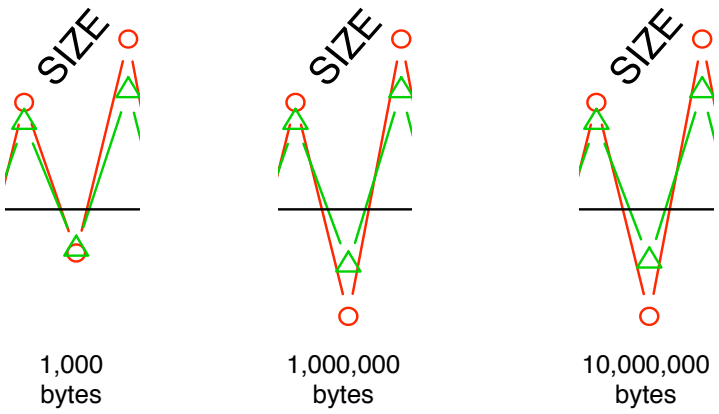


Figure 7.4: Close-up of `SIZE` messages from the sequence plot for transfers of three different file sizes.

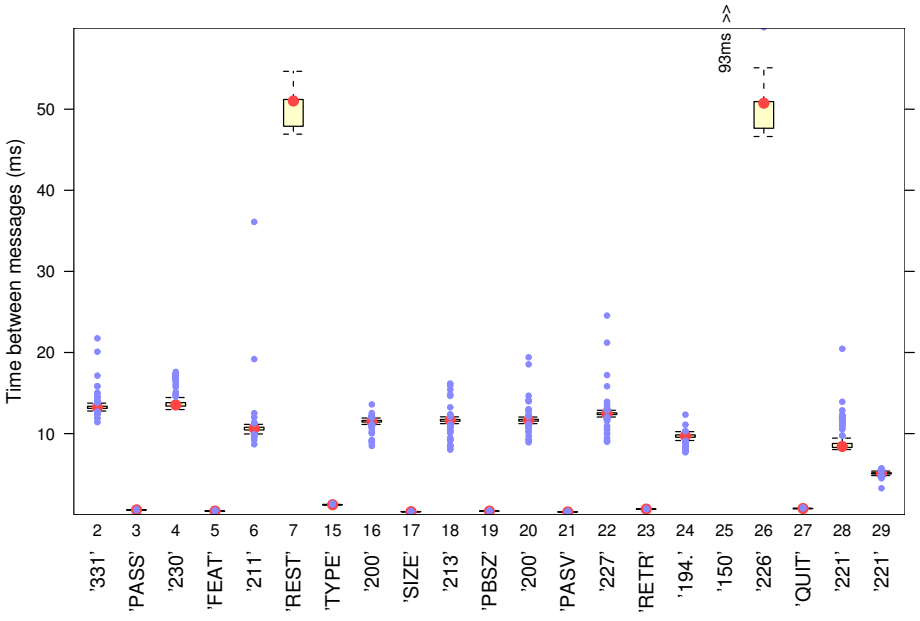
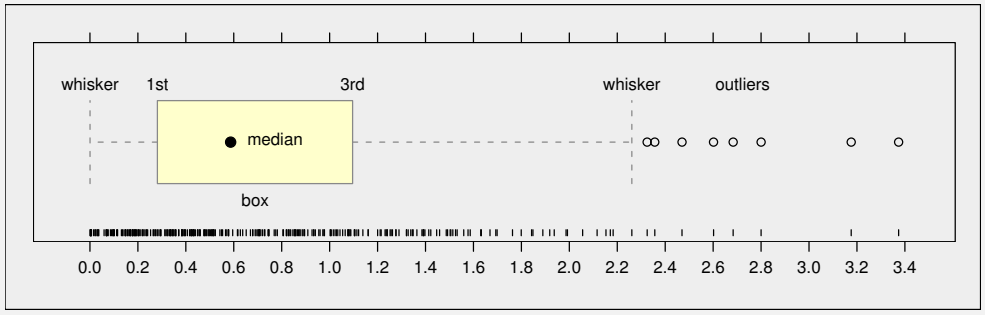


Figure 7.5: Box and whiskers plot showing delay distribution for each GridFTP message aggregated from 10 runs each hour for one day. Red spot indicates the median.

Box and whiskers plots

A 'box and whiskers' plot is a simple way of summarising the distribution of a data set. Firstly, it depicts the median and the interquartile range (IQR), which is determined by the value of the first and third quartiles (equivalent to the 25 and 75 percentiles). The IQR is represented by the box in the plot. The median is always inside the box.

The plot below describes 300 samples of the form $|x|^{1.1}$, where x values are drawn from the normal distribution ($\mu = 0, \sigma = 1$). The discrete sample values have been added to this diagram as small tickmarks just above the x-axis to aid in the interpretation.



The 'whiskers' of the plot extend to the most extreme data point which is no more than 1.5 times the IQR away from the box. Here, the IQR is about 0.8, and since there are no values less than 0 the lower whisker aligns with the smallest sample value. The higher whisker, however, aligns with the sample at around 2.25. Samples beyond the whisker are drawn as outliers.

The box can be used to evaluate the spread of the data set, and the relative positions of the whiskers with respect to the box indicate the degree of skew present (this example is skewed by construction).

Insert 7.1: Introduction to box and whiskers plots.

The timing for all 240 runs[4] per transfer configuration were aggregated to investigate the variance in the timings across multiple runs and during the course of a day. Figure 7.5 is a 'box and whiskers' plot[5] of the message timing for a transfer with default options. The y-axis values represent the time between the indicated message and the one before it. Messages arriving at precisely the same time (i.e., having a y value of 0) have been omitted, as can be seen between sequence numbers 7 and 15.

In general, the messages from client (`PASS`, `FEAT`, `TYPE`, etc) show a negligible time delta due to the network trace having been taken using `tcpdump` on the client machine. Most of the server messages have a median near 10 ms, which was approximately the RTT for the route between client and server. However, the `REST` and `226` replies from the server have a distinctly different profile with me-

4. 24 hours × 10 runs per hour
5. Insert 7.1 explains box and whiskers plots.

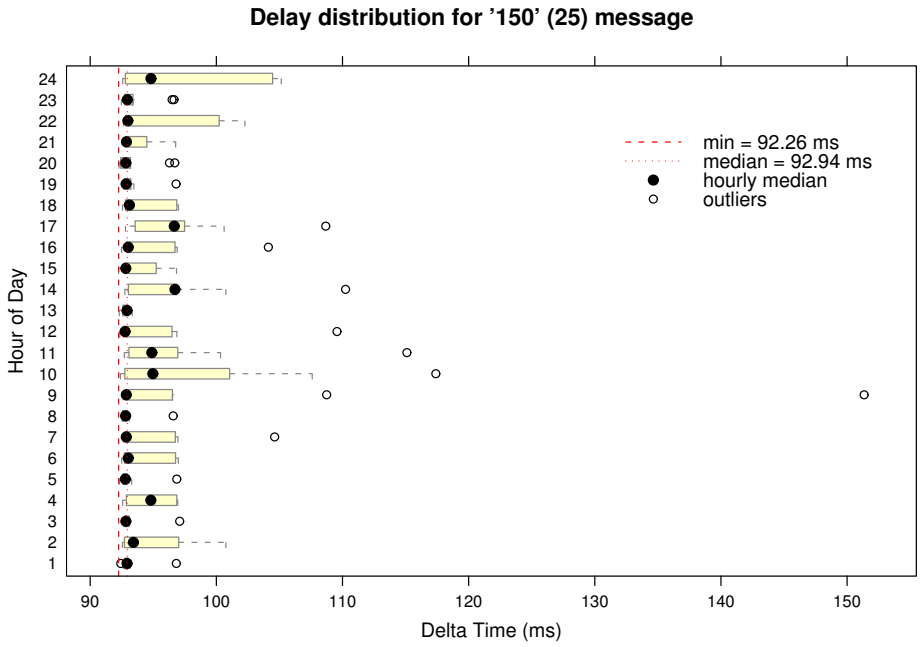


Figure 7.6: Box and whiskers plot showing delay distribution by hour for GridFTP `150` messages corresponding to column 25 in Figure 7.5.

dian around 50 ms. These timings are an artefact of the TCP Nagle [79] algorithm, where a second segment from server to client (sequence number 7) is delayed until an ACK is received for the first segment (6). The sending of the ACK itself is delayed by the receiver (client) in the hope that the ACK may be piggybacked on a data segment. Since the client sends nothing at this stage (it is waiting for the full server response), the TCP stack times out and transmits an ACK with no data.

Ignoring those deltas influenced by TCP stack behaviour, the remaining delta of interest is that at sequence number 25, the values of which (around 90 ms) are beyond the scale of the graph. This message corresponds to the acknowledgement by the server of the auxiliary data connection establishment (the `150` message), and is the main candidate for leveraging timing information to inform the real-time analysis of a GridFTP protocol conversation.

Figure 7.6 is a box and whiskers plot showing the delay distribution for these messages over the course of the day. It is clear that the timing has a minimum near 90 ms and there are more outliers during the working hours of the day than at other times, however the IQR and upper whiskers are not significantly different from others.

In order to make a check for any additional artefacts in the timing data, a quantile-quantile plot[6] was made of the message time deltas from every run against a normal distribution. The chart is shown in Figure 7.7. Mostly the measurements roughly adhere to the normal distribution, as would be expected from

6. Insert 7.2 on page 117 describes quantile-quantile plots.

random measurement error. Some messages show deviations from the straight line that indicate a heavy-tailed distribution. Other messages show a bimodal distribution (e.g., `REST`, `226` and `221`). The distribution in the `REST` messages is due to the Nagle algorithm described above, and the `226` and `221` differences appear to be the result of a process scheduling quantum on the server machine.

7.4.4 Discussion

The preceding investigation examined the GridFTP protocol structure in order to establish whether the size/timing properties seen at the transport flow level exhibit any regular patterns that may be used to more accurately classify bulk flows as GridFTP. It was shown that there is a regular structure (as would be expected from the use of a fixed client application) and that the creation of a data connection corresponds to a significantly longer inter-control message delay (see element 25 from Figure 7.5) than the normal request-response sequence.

However, since following such a delay is a period of inactivity during the actual transfer, a simpler approach presents itself: tag new bulk connections as 'probably' Grid if a bulk flow is established during the seconds after the last control message. Figure 7.8 depicts the coarse timing relationship between activity on the GridFTP control session and the start of corresponding data flows. The phase of a control session can be determined by observing this activity. The authentication handshake is identifiable from its content (FTP `ADAT`—authentication data—commands are used instead of `ENC`). Thereafter is the 'Initial commands' phase, the end of which is determined by the start of the 'Bulk data establishment' phase. It begins with the `RETR`[7] command and finishes with the `150` response. Although neither of these messages can be identified directly because they are encrypted, the control connection inactivity in the following 'Bulk data phase' means that the last two messages after a timeout correspond to 'Bulk data establishment'. The 'Final commands' phase is indicated by subsequent control connection activity.

Only connections established during the 'Bulk data establishment' phase are eligible to be treated as GridFTP bulk data. The time window should be small (several RTTs) which limits the possibility of a false positive. Note in Figure 7.8 that bulk data connection 2 is shown as starting inside the window, whereas connection '... n' starts outside. In fact, only the first connection is guaranteed by the protocol to be inside the window; the timing of subsequent connections is dependent on how quickly they can be established. Despite this discrepancy, connections 2 and beyond can still be identified, since the destination TCP port number is the same as for connection 1.

Before describing and evaluating an implementation using the timing information in the following sections, the next paragraph considers the possibilities for

7. Where `RETR` appears hereinafter, `STOR` is also possible.

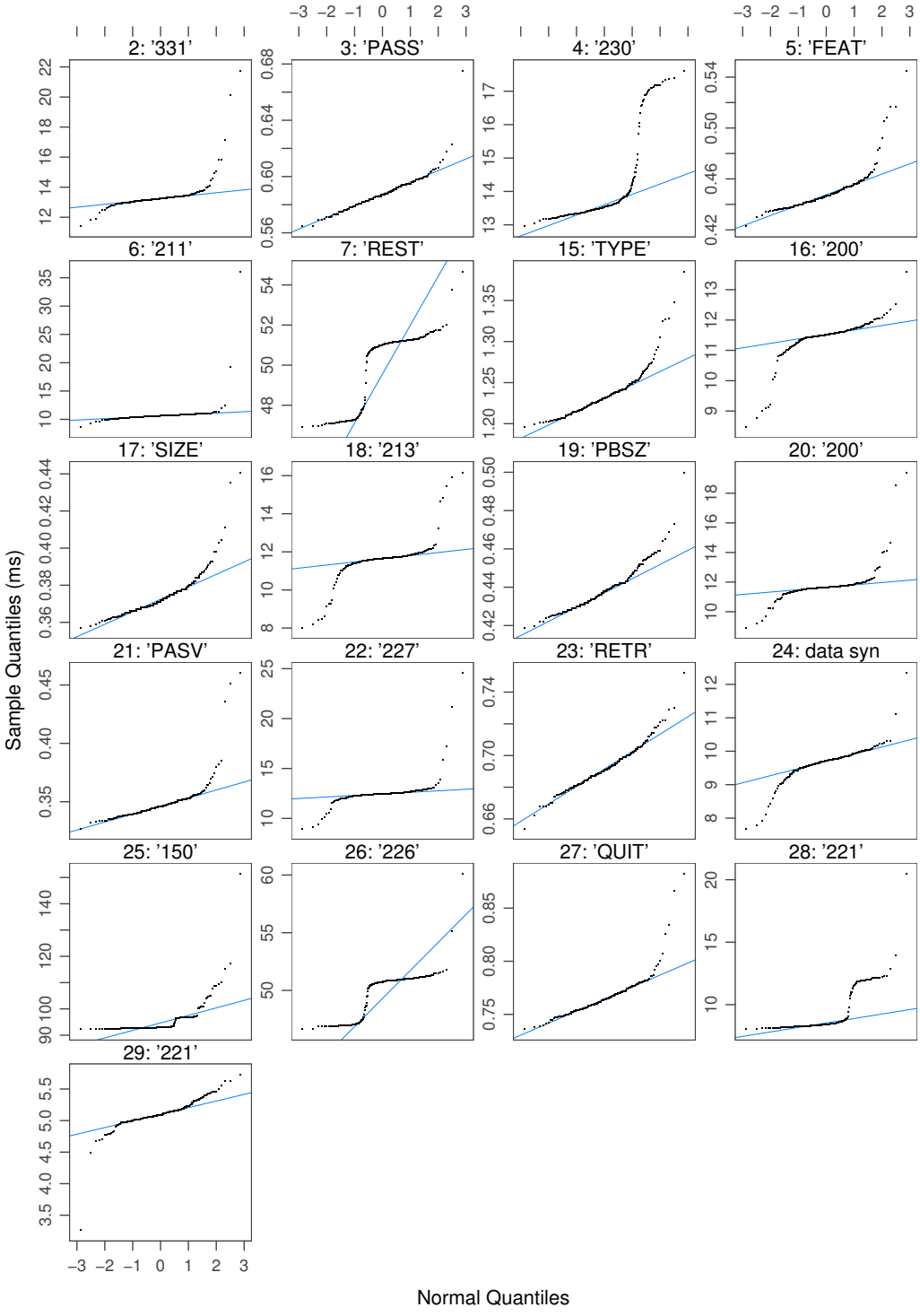
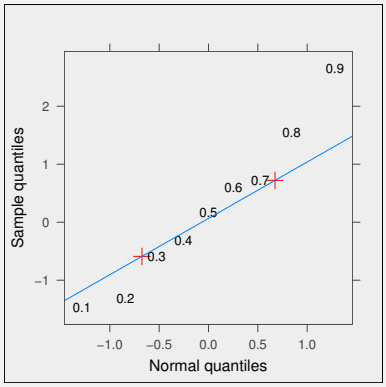


Figure 7.7: Quantile-quantile plot of time between encrypted GridFTP messages. Above each plot is a sequence number followed by the initial message content.

Quantile-quantile plots

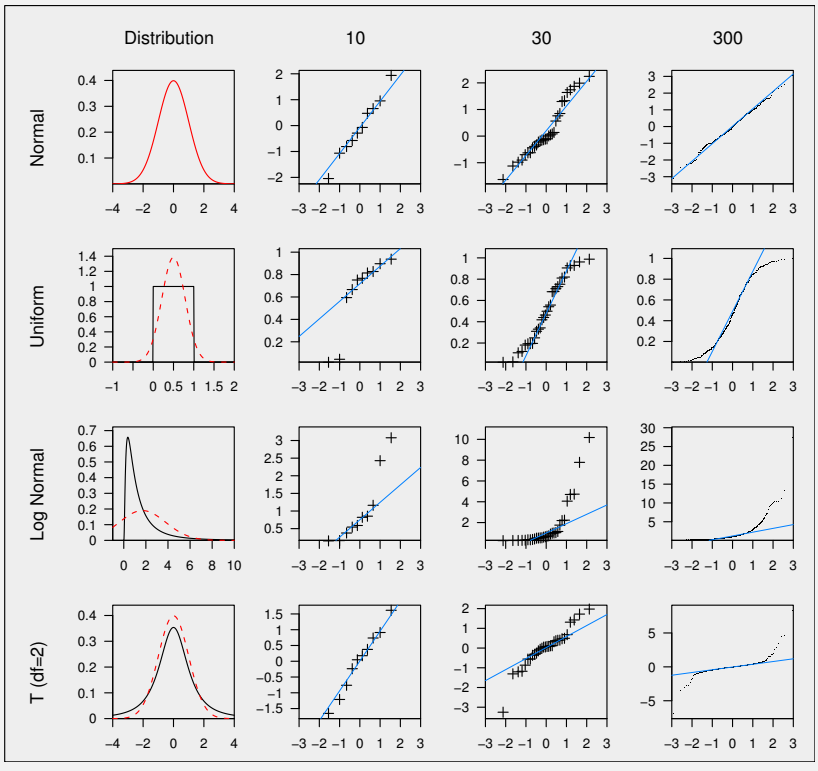
A 'quantile-quantile' plot is a graphical way of comparing the distribution of two sample sets. When one of the sets is drawn from a known distribution, the plot provides a way to evaluate whether the unknown data are likely to be distributed in the same way.

A simple Q-Q plot could be formed by evaluating the 0.1, 0.2, ..., 0.9 quantiles for the unknown and known data sets, then plotting then against one another. The plot below shows this, where points are represented by the quantiles. Notice that the x-axis value for the 0.5 quantile is 0, since this is the median of the standard normal distribution (mean 0, standard deviation 1).



The red crosses indicate the 1st (0.25) and 3rd (0.75) quartiles (quantiles), through which a straight line is drawn. Notice how the crosses do not correspond to any actual data point in this graph, because the nearest values are 0.2/0.3 and 0.7/0.8.

Since this plot is based on very few data points, limited inferences can be drawn from it. With more sample points, however, the plot can be used to identify departures from normality (or whatever other distribution is of interest). Data sets drawn from the same distribution should yield points falling roughly along the straight line. Departures from the line indicate skewness, heavy or light tails, and possible extreme values.



The table of charts above shows four distributions (normal, uniform, log-normal and t) and sample Q-Q plots for sample sizes 10, 30 and 300.

Insert 7.2: Interpreting quantile-quantile plots.

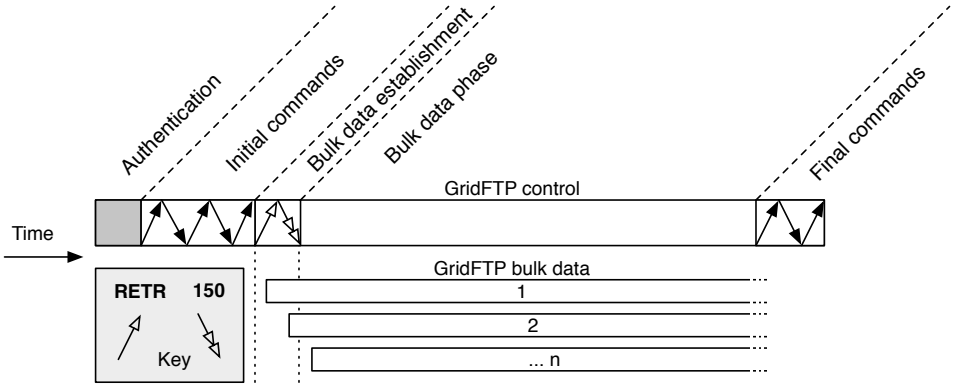


Figure 7.8: Timing relationship between GridFTP control connection phases and data connection establishment.

gathering additional transfer information (question Q3 from Section 7.1).

At the end of Section 7.4.2 on page 110 the relationship was investigated between plain-text message lengths and the encrypted message lengths that have been subject to padding. It was shown that differences in the response to a `SIZE` command were sometimes visible (in the same way that changes in the file name length of a requested file were apparent). Unfortunately, the coarseness of the padding (16 bytes in this instance) means that there is only one switch-over point between a short and long encrypted message, and therefore it would only ever be possible to differentiate between file sizes of more or less than some threshold[8]. In the example given the threshold was between 1,000 and 1,000,000 bytes. A further complication is the correct identification of the `SIZE` message from just observing the encrypted control messages, which really depends on a fixed command sequence used by the client. Although the equal lengths of `SIZE` and `RETR` messages could be used to inform this detection, the ultimate benefits are limited. Hence, the focus is on the correct identification of bulk data flows themselves.

7.5 Simplified GridFTP Classification

GridFTP control sessions can be classified by inspecting the payload content, since they are based on plain FTP but use GSSAPI security services, and also because a server usually responds with a distinctive message[9]:

```
220 tim GridFTP Server 1.17 CAS/SAML enabled GSSAPI type Globus...
```

Therefore, the identification of the application (Q1) is guaranteed.

To recap on the approach: the lifetime of a control session between a pair of hosts gives a coarse time window within which new connections between a partic-

8. A power-of-10 threshold representing a change in the number of decimal digits transmitted to represent a file size.
9. The classification still works if the server administrator has disabled the banner.

ular host pair become candidates for classification as GridFTP bulk data. By using knowledge of the control protocol (as discussed in the previous section) the time window can be reduced, and consequently the false positive rate improves.

In addition to the time-based argument previously introduced, some configurations of a GridFTP file transfer exhibit a further property that can be used to definitively identify a corresponding bulk data session. Recall that the GridFTP control session uses GSI to negotiate certificates for authentication. If data channel authentication is enabled (via an FTP command issued during the 'Initial commands' phase), then the data connections also use GSI to authenticate. The purpose is to prevent an attacker connecting to client or server (depending on whether active or passive transfer mode has been enabled) and intercepting the data. The key insight here is that the data connections then look like a TLS/SSL session (remember that GSI is layered on top of TLS/SSL), and will negotiate *the same certificates* as were used on the control session. Therefore, a second, more robust, way of identifying GridFTP bulk data is to correlate certificates between data and control sessions.

7.5.1 Design

So, there are two techniques to classify bulk flows as GridFTP:

1. Correlate connection start times with control session phases.
2. Correlate TLS/SSL certificate exchanges with control session certificates.

The first of these is combined with an additional method to determine whether a connection is likely to be carrying bulk data.

Figure 7.9 depicts the relationship between different sets of TCP connections that are candidates for classification as GridFTP bulk data. The first data connection and subsequent connections are shown separately, since their timing restrictions are different. Ignoring the DCAU/SSL segments for the moment, the first data connection is one that is established during the `RETR/150` time window and has 'consistently large packets' (the definition of which will be given shortly). For subsequent connections the time window is relaxed to 'soon after the `150` response', provided the destination port number matches the first connection.

Connections using data channel authentication are found by identifying the SSL content and matching certificate usage with the control connection. The `RETR/150` time window and packet-size restrictions are not enforced here both because the certificate match is accurate and the SSL protocol messages would be unlikely to meet the packet-size criteria. Furthermore, a certificate match disables the time-based heuristic approach for the control connection in order to reduce the risk of false positives.

The technique for identifying 'consistently large packets' is simple: the number of data-bearing packets on a flow is counted, along with the total data amount.

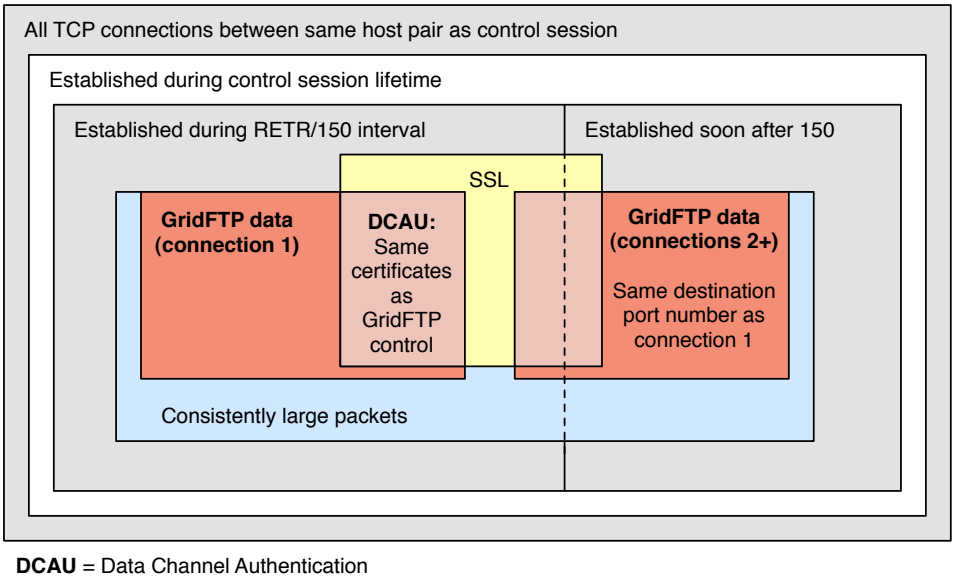


Figure 7.9: Sets of potential GridFTP data connections.

The average packet size can then be calculated and compared with the maximum segment size for the link. If it is within a reasonable threshold (indicating mostly maximum-size packets) then the flow is likely to be transporting bulk data of some sort. A further restriction is also added: the flow must transmit data in only one direction. This simple addition ensures that bidirectional connections—such as SMTP—will never be matched.

As described in Section 7.4.4, in order to identify the `RETR/150` time window, it is necessary to wait for the GridFTP control flow to become idle. The idle timeout needs to be long enough to not be triggered by the time between messages in the 'Initial commands' phase (see Figure 7.8). However, the length of the timeout also influences the response time for the detector. Since it is necessary to wait for the connection to be idle before identifying the non-DCAU bulk flows, data may go undetected for a short period.

7.5.2 Implementation

The time-window and SSL-certificate-based GridFTP bulk data identification approach has been implemented in the real-time application protocol analyser introduced in Chapter 5. Two new analyser classes were added to support the scheme; these were the GridFTPAnalyser and SSLAnalyser. The purpose of the latter is to identify SSL connections and compare any certificates exchanged with those found on a GridFTP control session between the same host pair. It does not report any events directly—it just delegates to the BulkDataAnalyser.

The GridFTPAnalyser is triggered by the FTP matching module mentioned in Section 6.3.5. It processes the authentication exchange and stores a hash of each

```
# After a short period of inactivity on GridFTP control
bulk_connections = []
window = last_client_msg_time .. last_server_msg_time

# First check for bulk connection starting in the window
for c in connections_between_hosts(src,dest) do
  if  c.start_time in window and
      c.mss - c.average_packet_size < THRESHOLD and
      c.data_in_only_one_direction? and
      not c.known_protocol? then
    c.assign_protocol_analyser BulkDataAnalyser("GridFTP-Window")
    bulk_connections.append(c)

    # Match forthcoming connections to same host/port
    register_future_connections c.dst,c.dst_port,
                                BulkDataAnalyser("GridFTP-Port")
  end
end

# Now match against existing connections to same destination port
for b in bulk_connections do
  for c in connections_between_hosts(src,dest) do
    if b != c and b.dst_port == c.dst_port then
      c.assign_protocol_analyser BulkDataAnalyser("GridFTP-Port")
    end
  end
end
```

Listing 7.1: GridFTP heuristic analyser pseudocode.

certificate used for later interrogation by the SSLAnalyser. It also keeps track of the time of each request and response on the connection and schedules a timer to trigger the search for bulk data connections according to the process described in Section 7.5.1. Since the granularity of the timers provided by the real-time monitor is one second, the timeout period can only be a whole number of seconds, and may be scheduled up to a second later than expected due to the calendar queue timer implementation (see Section 5.4). Listing 7.1 shows pseudocode for the identification process.

A threshold value is shown in the listing for the 'consistently large packet' criteria, which represents the allowable deviation, on average, from the connection MSS (Maximum Segment Size). The threshold was set to 30 bytes. This is intended to cover a moderate number of TCP options (which reduce the available space for true data payload). The MSS is extracted from options in the initial TCP SYN handshake, or else assumed to be around 1460 bytes (based on a standard Ethernet MTU of 1500 bytes).

7.5.3 New Events

The new analyser introduces some new events and extends the functionality of some existing events. For more information on the event system, see Section 6.3.1.

Generic Bulk Data

The generic bulk data analyser was extended to support the new usage within the GridFTP analyser:

`BulkData:Opened(...,orig_bytes,resp_bytes,elapsed)`
: In addition to the existing parameters described in Section 6.3.2, when the analyser is attached to a flow after it has been established, the number of bytes already passed on the flow is recorded along with the elapsed time. This supports the determination of the number of bytes that were missed before the flow was identified.

GridFTP Analyser

When the GridFTP analyser identifies a bulk data connection, it registers a BulkDataAnalyser in the same way as the plain text protocol analyser described in Section 6.3. However, it makes special use of the `source` field in order to differentiate between identification mechanisms:

`GridFTP-SSL` Identified from SSL certificate match.

`GridFTP-Window` Flow started inside the `RETR/150` window.

`GridFTP-Port` Same destination TCP port as a `GridFTP-Window` flow.

The following specific events are generated by the GridFTP analyser:

`GridFTP:Control()`
: Registers the start of a GridFTP session.

`GridFTP:Closed(stats,window)`
: Summarises the results of the GridFTP analyser. The `window` parameter indicates the duration of the `RETR/150` window.

  The parameter `stats` contains connection counts for the following criteria:

  - SSL certificate match (`GridFTP-SSL`).
  - Accepted inside `RETR/150` window (`GridFTP-Window`).
  - Accepted due to same destination port as an accepted connection (`GridFTP-Port`).
  - Inside `RETR/150` window but rejected due to not bulk data.

  These values are a useful diagnostic for assessing the accuracy of the detector.

7.5.4 Real-time Monitor Changes

The implementation described above required some isolated changes to the monitoring system. These were easy to implement, and did not require any modification to the existing analysers.

Host-pair connection lists As the `connections_between_hosts` method from the pseudocode in Listing 7.1 suggests, it is necessary to be able to efficiently enumerate the active connections between a source-destination IP address pair. The existing connection table was a single hash table indexed by the TCP connection four-tuple. An auxiliary table was introduced, keyed by host-pair. This required the addition of an extra field in the TCP connection record to maintain a doubly linked list.

Connections are not added to this table until they have been fully established. When added, the list is guaranteed to be kept in order of connection start time, which means that an analyser making repeated passes over the list can do so incrementally by keeping track of the most recent connection visited.

Maximum Segment Size parsing The TCP option for negotiating the MSS during the three-way handshake is parsed, and the value stored in the flow record.

Packet count The TCP flow manager previously kept track of the number of bytes observed on a flow but not the number of packets. This was added.

The memory costs of the above changes are an additional 20 bytes (approximately 10%) per connection record (8 bytes for linked list entry plus a per-flow 32-bit packet counter and 16-bit MSS field). The secondary host-pair connection table incurs a processing overhead when connections are established or freed, but otherwise the only CPU costs are when the table is traversed by the GridFTPAnalyser. Note that the table update takes place upon completion of the three-way handshake, and therefore the system performance when coping with SYN flooding (as covered in Section 5.5.1) is unaffected.

The per-connection costs for SSL connections (as used by HTTPS, secure IMAP, etc) increase due to the requirement to look for an associated GridFTP session. However, the lookup is only necessary if both client and server certificates are exchanged, and most SSL connections involve only a server certificate.

7.6 Evaluation

An experiment was carried out to examine the effectiveness of the simple GridFTP data classification system. Three hosts were used; one client machine (`nsmc06`) at Glasgow University was set up to perform GridFTP transfers to and from two other machines. These were called `bananarama` and `kosciusko`. The former machine

was also used in the experiments from Section 6.5.5, and is located at Lancaster University. The latter machine, `kosciusko`, is at Glasgow University and the network topology is such that communication between `nsmc06` and `kosciusko` passes the monitoring point[10].

7.6.1 Method

A script was developed to run the `globus-url-copy` command-line file transfer program with a range of options, host source/destination and file sizes. The most important options were those controlling the use of data channel authentication and the number of parallel TCP streams to be used. Transfers were performed with either one stream or four. The file sizes used were 10, 25, 50 and 100 Mbytes. Although these file sizes are much smaller than might be expected to be transferred using GridFTP, they represent a sufficient range to demonstrate the performance of the monitoring system.

The script was configured to run a random[11] series of file transfers between `nsmc06` and one of the other two machines (transferring files in either direction) over a 24 hour period. Between each transfer was a random idle period of between 30 and 90 s.

In order to ensure that there were additional TCP connections between the hosts that might trigger a false positive, just before starting each file transfer a process was started to produce a set of 'confounding' connections, with the following properties:

- 20 connections intended to be non-bulk
- 5 bulk connections transporting 1 Mbyte
- Two bulk connections transporting up to 50 Mbytes at random

The connections from the first two items were each started with a random delay of up to four seconds. The two $random(50)$ Mbytes connections were started with a 400 ms offset. These connection counts, timings and transfer sizes were empirically chosen to offer a reasonable chance of meeting the time-window criteria, and therefore triggering false positives. The non-bulk connections were intended to test the effectiveness of the large average packet measurement. They were represented by a process shown in Listing 7.2. The bulk and non-bulk connections were directed towards different fixed ports on the target machines in order that the connections could separated during the results analysis.

During the course of the experiment, the real-time monitor was reporting on the network activity. The timeout period (see Section 7.5.2) was configured to be the minimum of one second.

10. Point B from Figure 3.1 on page 32.
11. Each random value here was picked from the `rand` method provided by the Ruby scripting language, which, according to the documentation, is a 'modified Mersenne Twister with a period of $2^{19937} - 1$'.

```
# Connect to server socket
s = TCPSocket.new(@host, NON_BULK_PORT)
400.times do
  # Write 1 to 1000 bytes
  s.write("N" * (1+rand(1000)))
  # Get random number between 0 and 1
  r = rand
  # Sleep up to 2 seconds, biased towards shorter durations
  sleep (r*r * 2)
end
```

Listing 7.2: Non-bulk simulation connection code.

7.6.2 Results

Every GridFTP bulk-data connection using data-channel authentication (i.e., the SSL certificate match is possible) was correctly identified (there were 804 such connections), and there were no false positives. Since a certificate-based match disables the heuristic time-based analysis, this is as expected. Furthermore, since the matching is triggered by the certificate exchange on the data connection before the bulk-data phase starts, it is possible to properly report the entire bulk data connection.

The remainder of this section presents the data showing the efficacy of the time-window-based analysis. Figure 7.10 shows the distribution of the duration of the `RETR/150` time window for the two target hosts. The closer machine, with a round-trip time of approximately 4 ms, has a median window duration of 28 ms, whereas the median duration of the remote machine (RTT $=$ 12 ms) is 55 ms. The latter measurement has a greater spread, as would be expected with the traffic being subject to more Internet routing delays and losses than the local machine. Even so, the window size is still small in both cases; certainly it is usually less than 150 ms ($\Pr[\mathit{window} < 150\,\text{ms}]$ is 96% for `bananarama` and 99% for `kosciusko`). Therefore, over the lifetime of a GridFTP transfer session (which could be several minutes for a large file), the time window during which false positives may be triggered is relatively very small.

Recall from Section 7.6.1 that files of size 10, 25, 50 and 100 Mbytes were used. All of the bulk data connections from the 25 Mbytes and higher transfers were correctly identified. Of the 10 Mbytes transfers, only 17% of the connections were matched. This is caused by the transfer lasting insufficiently long for the control session inactivity timer to trigger before the data has finished.

Of the connections that were successfully identified (including all of the 25 Mbytes and larger files), some of the data was transferred before the analyser was able to report on it. This is related to the inactivity timer, and the volume of data missed is proportional to the timer duration and the connection throughput in the first few seconds. The results mentioned in this and the preceding paragraph are summarised in Table 7.1. Note how the 'Missed Volume' percentage is inversely

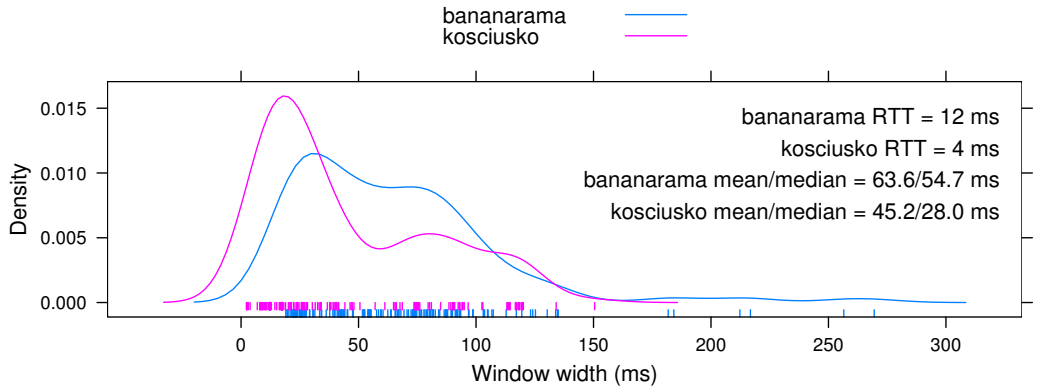


Figure 7.10: Density plot showing distribution of GridFTP RETR-150 time window durations. Marks along the baseline show the actual values.

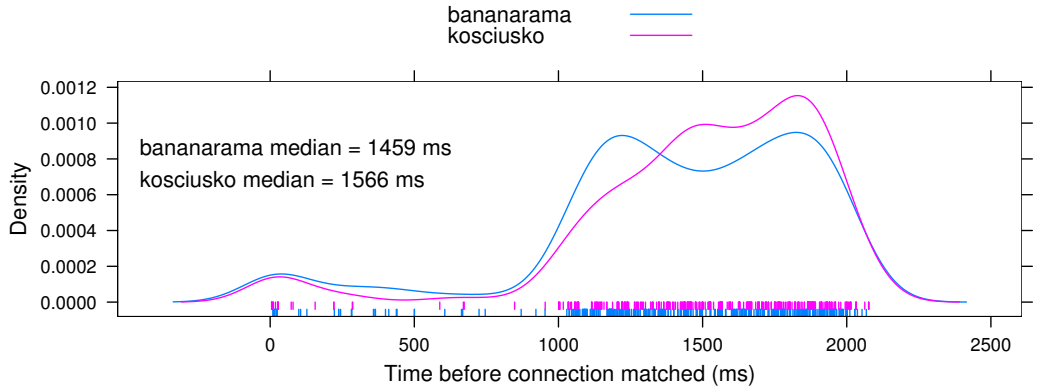


Figure 7.11: Density plot showing distribution of delay from connection start time until a GridFTP bulk data connection is identified. Marks along the baseline show the actual values.

Size (MBytes)	Total Transfers	Bulk Data Connections	Matched	Missed Volume	False Positives	
10M	116	320	17%	68.8%	3	2.6%
25M	93	225	100%	46.9%	16	17.2%
50M	103	247	100%	22.1%	10	9.7%
100M	100	250	100%	11.5%	8	8.0%

Table 7.1: Connection counts by file size for the non-certificate-based matcher, showing total number of transfers of each size, the number of resulting data connections and the proportion of these connections correctly identified by the analyser. 'Missed Volume' column represents the proportion data transferred on a connection prior to being identified by the analyser. 'False positives' gives the number of transfers that triggered a mis-classification.

proportional to the transfer size (since the available bandwidth is fixed) and that number of 'Bulk Data Connections' is approximately 2.5 times the number of transfers. This is because the transfer configuration was randomly chosen between one or four data connections per transfer (the average of the total is 2.5).

Table 7.1 shows a 'False Positives' column, which indicates the number of transfers for which a bulk data connection was incorrectly identified (i.e., a connection started during the `RETR/150` time window that was not GridFTP bulk data). The consequences of these results are considered shortly in Section 7.6.3.

The distribution of elapsed times before a bulk data connection is classified was investigated, and is shown in a density plot in Figure 7.11. The values are mostly uniformly distributed between one and two seconds, which corresponds to the expected behaviour of the coarse-grained timer implementation. The measurements below one second represent connections classified by having the same destination port as another bulk data flow; these are not dependent on the timer interval[12].

7.6.3 Discussion

Section 7.6.2 presented the raw data resulting from the experiment. Recall that 100% accuracy and no false positives was achieved for the certificate-based approach. For the heuristic approach, files of size 25 Mbytes or larger were identified correctly (no false negatives), showing that the technique is effective for reasonably sized file transfers. However, some of the bulk-data 'confounding' connections described in Section 7.6.1 triggered a false positive match (none of the non-bulk connections matched). These are considered next.

It is difficult to give a specific value to the false positive rate for the technique, because it is strongly dependent on the type of traffic present between the host pair. Bear in mind that the 'confounding' connections were designed explicitly to pass the criteria used by the heuristic technique. Only traffic with the following properties is a potential match:

1. Average packet size in first few seconds close to MSS.
2. Unidirectional.
3. Start within `RETR/150` time window (typically smaller than 150 ms).

In particular, any connection used by common Internet protocols (such as SMTP or HTTP) will never match because they are bidirectional. The average-packet-size restriction was evaluated independently on SMTP connections from a week-long trace at the DCS monitoring point. Of 110,901 email transmissions, only 59 (0.05%) matched after one second.

Given the above three restrictions, the likelihood of a connection being misclassified is extremely low. Any connection that did cause a false positive would

12. They correspond to pseudocode line 15 or lines 21 and later in Listing 7.1

have to have bulk-data-like properties, and therefore its treatment as GridFTP-bulk by a management system would hopefully not be too problematic. An additional heuristic could be implemented to identify false-positives at the end of a GridFTP transfer (see Section 8.2.2), therefore providing valuable information to the management system on the system accuracy.

For the connections that are identified correctly, the timeout period means some initial amount of data will pass before the monitoring system is notified. However, relative to the total transfer size for large files the missed data is insignificant. For example, consider the case of a wide-area Internet path with 200–250 ms RTT and (theoretical) infinite link capacity. The amount of data transmitted within two seconds assuming the slow start behaviour of TCP Reno [88] and 1460 byte MSS can be estimated as follows. In cycle i (counting from 0), 1460×2^i bytes will be transmitted. Therefore, after n cycles, the total number of bytes transmitted is $1460 \times \sum_{i=0}^{n-1} 2^i = 1460 \times (2^n - 1)$. 200–250 ms RTTs within a two second interval corresponds to 8–10 cycles (neglecting transmission times due to the infinite link capacity), and therefore between 365 and 1460 Kbytes. With jumbo Ethernet frames of up to 9000 bytes the upper limit increases to around 9 Mbytes.

Improving the immediacy of event-reporting would mean reducing the timeout interval. This, in turn, would require potentially expensive modifications to the real-time monitor to support more fine-grained timer activity. As it stands, the one-second timeout appears to work well, but may need to be extended for file transfers between more distant hosts than those involved in the experiment given. There is potential for determining the timeout dynamically based on the observed timing of the 'Initial commands' phase (see Figure 7.8).

To answer the questions posed in Section 7.1 for the case of GridFTP:

1. *Can control flows still be classified according to the application in use?*
   Yes, reliably.
2. *Can data flows be matched with their corresponding control flow?*
   100% accuracy for transfers with data channel authentication.
   100% true positive rate for transfers above a threshold size, depending on throughput (e.g., around 25 Mbytes). Negligible false positive rate for common Internet protocols.
3. *What additional information can be extracted?*
   File sizes at end of transfer. Whether data channel authentication is being used. Number of parallel transfers.

In summary, a protocol analyser for encrypted GridFTP traffic that achieves very high accuracy has been constructed, based on a detailed investigation of the protocol behaviour. Minimal changes were made to the real-time monitoring system (described in Chapter 5) in order to support this new technique, which complements the plain text analysis approach covered in Chapter 6. The next, and final,

chapter draws together the key results from the entire dissertation and examines areas for future work.

Chapter 8

Conclusion

The Thesis Statement of Chapter 1 presented three assertions, covering the building of a real-time network monitor for plain text protocol analysis, analysis of encrypted traffic, and generation of events to support traffic re-engineering. The first part of this final chapter covers each of these in detail, with reference to the completed work. The second part considers the areas for future work.

8.1 Validation of the Thesis Statement

8.1.1 Real-Time Network Monitor and Plain Text Analysis

T1. *A real-time network monitoring tool can be built that performs full plain text analysis of Grid bulk data traffic at line rate.*

Chapter 5 presented the detailed design, implementation and evaluation of a Gigabit Ethernet real-time monitoring system for analysis of application-level protocols. Three novel aspects of the design ensured that the system was able to meet the requirements: the use of the 'retained packets' scheme (Section 5.3.2) to avoid memory copies and buffering wherever possible; the ability to efficiently skip bulk-data phases of a mixed control/data flow (Section 5.3.5); and the use of a co-routine-based threading library for the efficient and easy implementation of protocol analysers (Section 5.4.1).

The monitoring system supports full reassembly of TCP flows and classification of application protocols by initial string matching rather than well-known ports. Protocol analysers may be implemented at a low level (operating directly on raw packets or a reassembled TCP stream) or be based upon the threading library mentioned above. The use of unit testing and regression testing against a library of trace files increases confidence in the reliability of the system.

The system was shown to cope with high rates of new connections per second (400,000 packets per second or more), as might be experienced under a SYN-flooding denial-of-service attack (Section 5.5.1).

Chapter 6 built upon this monitoring system to implement several protocol analysers dealing with the plain text content of the applications introduced in

Chapter 4. A case study of analyser development (Section 6.5.4) was presented for `iperf` in order to demonstrate the ease with which analyser modules can be implemented.

The plain text analysers were able to accurately identify the protocols in question and extract from them information about auxiliary or embedded bulk data flows. A blind test experiment was carried out to validate these claims (Section 6.5.5). It showed that, except in the unusual case identified in that section, the analysers had 100% accuracy in identifying bulk data connections, and in many cases that transfer size information was available in advance.

Section 6.4 presented results from applying the monitor to extract information about HTTP and FTP usage from a week-long trace, demonstrating the flexibility of the system and the quality of the information obtained.

Based on a trace-driven timing analysis of HTTP connections, the monitor was shown to operate at speeds approaching 10 Gbits/s. While this result would be somewhat reduced in real-time operation due to additional contention for the memory bus, it is a significant result that validates the design of the system and indicates the potential for scaling to faster networks.

8.1.2 Heuristic Analysis of Encrypted Traffic

T2. *This tool can be extended to perform heuristic analysis of encrypted or inaccessible payloads at line-rate.*

Chapter 7, building upon the work of the previous two chapters, covered the possibilities for leveraging information available at the network monitor for reporting on encrypted GridFTP file transfers. Theoretical approaches to extract additional information were considered, and then Section 7.4 presented a detailed investigation of the message time and size properties from a range of GridFTP sessions. Several observable features were identified (such as the message size relationship of file path lengths and transfer sizes, and the distinctive control channel timing profile at the start of bulk data transfers). These results were then used in Section 7.5 to construct a dual-approach system for GridFTP transfer identification.

The system was implemented in the real-time monitor described in Chapter 5 and previously used in the work of Chapter 6. Although a small number of changes were required to the core of the monitor, these were easy to implement and caused minimal or no disruption to the monitor and other protocol analysers (Section 7.5.4).

An evaluation of the new protocol analyser was carried out (Section 7.6), which yielded 100% accuracy with no false positives for the certificate-based approach. The 'fuzzier' time-window-based approach correctly identified all bulk transfers of 25 Mbytes or more within two seconds, but was susceptible to some false positives. Section 7.6.3 argued both that the real-world likelihood of such false

positives was negligible and that any false positives would be unlikely to be mistreated by a network management policy.

8.1.3 Event Generation for Traffic Re-engineering

T3. *Such analysis can generate events early enough to enable dynamic traffic re-engineering.*

The previous two sections have dealt with the evidence to support the development of a real-time monitor and the analysis of both plain text and encrypted control traffic. The implemented systems generate events (as described in detail in Sections 6.3 and 7.5.3) that may be interpreted by a network management system (or some intermediary) to dynamically re-engineer the corresponding TCP flows. As mentioned in Chapter 1, it is expected that this will be achieved by updating the rules for determining a Forwarding Equivalence Class in an ingress Label Switching Router.

The events corresponding to the plain text analysers of Chapter 6 are generated *before* the start of the corresponding TCP connection, and therefore it is possible for a rule update to be realised before any data is transported over the connection (depending, of course, on the specifics of the management system and its interface).

The encrypted analysis of GridFTP traffic identifies bulk data flows in one of two ways, and each has different timing properties. The certificate-based matching technique identifies flows after they have been established, but before they begin to transport bulk data. The time-window-based approach incurs up to a two second delay from the start of a bulk transfer connection before an event is reported, and therefore a quantity of data proportional to the connection throughput will have passed before the event is delivered.

Events can be delivered to a local process on the monitoring host (see Section 5.4.4) within 1 ms. From there the event may be processed further before activating some change (if policy dictates) elsewhere in the management system. Therefore, the event delivery introduces a negligible delay in the ability of the network management system to react to the event.

8.2 Future Work

There are several avenues for future development of the work presented in this dissertation. This final section examines possible directions to continue the research.

8.2.1 Real-Time Monitor

Further development of the real-time monitor of Chapter 5 would involve extension to faster line speeds, perhaps simply by swapping-in the next 'step-up' from the Gigabit Ethernet DAG card, the DAG4.3S for OC-48 links (with total full-duplex

bandwidth of 5 Gbits/s). User-level packet monitoring is limited at high bandwidths (such as 10 Gigabit Ethernet) by bus bandwidth. Therefore, the possibility of dynamically pushing down packet filtering to the monitoring hardware should be investigated (for example, by using the Endace DAG Coprocessor [40]). The packet filtering would have to be dynamically modified according to the traffic identified by the real-time monitoring system.

As indicated in Section 5.5.4, the system design extends itself to multiple processors (either on the same machine or within a cluster). The main process would remain single-threaded, and message-passing could be used between monitor instances to access and update shared state. The costs and benefits of this approach need to be evaluated.

The heuristic analysis from Chapter 7 required some minor changes to the monitor to support the enumeration of flows between host pairs (Section 7.5.4). The solution implemented was to use a hash table and linked list structure to maintain this information in addition to the original connection table, which was keyed/hashed by the connection 4-tuple. An alternative approach of using a two-level hash for all connections should be evaluated.

The current system is limited in some ways by the available memory in the monitor, because each flow requires the monitor to keep track of some state. Although a few 32-bit pointer assumptions were made, extension to 64-bit machines should not be too difficult. This would significantly increase the memory limit, which is currently set by the 4 Gbytes virtual memory size of a 32-bit machine.

8.2.2 Protocol Analysis

The selection of Grid-style applications in Chapter 4 was constrained to bulk-data transfer tools, and from that class of applications a representative set was chosen. The selection could be extended to cover more bulk-data transfer applications (especially if a particular site is known to be primarily using one tool), but also other types of Grid application (such as real-time multimedia, interactive visualisations or tightly coupled computations). Of course, the predicted benefit of further development on a particular application should be backed by evidence that it is a significant user of network bandwidth. Human inspection of network anomalies, using appropriate tools, could be used to guide this.

Similarly, as new versions of the chosen applications evolve, the protocol analysers may need to be modified slightly to accommodate differences in the application-level protocol. Logging output from the analysers can be used to identify the presence of unexpected protocol behaviour. Provided the changes are not too great—which would normally be the case to ensure backward compatibility—the maintenance task should not be too onerous.

As indicated in Section 3.1, it was not feasible to place the monitor system at the edge of the Glasgow University campus network. However, discussions with

the Computing Service have indicated that they would be interested in deploying such a system once it is ready for operational use (rather than being under active development). It would be interesting to evaluate the system at this new location to assess its utility for monitoring at the institution level.

The heuristic analysis techniques presented in Chapter 7 focused on the GridFTP application. Similar techniques could be applied to other protocols, such as SRB with SEA encryption, that are identifiable from content inspection but whose main conversation is encrypted.

In the specific case of the GridFTP analyser, the protocol-specific timing patterns of the data connection establishment (see first paragraph of Section 7.4.4) could possibly be leveraged to eliminate the need for the coarse 2 s latency for bulk flow identification.

Section 7.3 restricted the heuristic analysis of GridFTP transfers to eliminate cases of third-party transfers. The certificate-based identification approach extends to third party transfers without difficulty, simply by relaxing the host-based constraint to include only the server host rather than both client and server. The time-window-based technique suffers more because of the need to enumerate a potentially greater number of connections due to the relaxed criteria. This in turn may lead to an increased chance of false positives. Whether this is a serious limitation depends on the typical ways in which a particular project uses the GridFTP system. For example, if a third party transfer is used remotely (i.e., across the monitored link) to transfer data between machines at a remote site, the bulk data flows will not pass the monitor point, and therefore not be cause for concern.

As suggested in Section 7.6.3, an additional heuristic could be implemented for identifying false positives at the end of a file transfer. Provided that the network operator's policy were to re-engineer rather than block an identified flow, a simple approach would be to use the observation that the parallel TCP streams used for file transfer should all transfer roughly the same amount of data and have a similar lifetime. Any incorrectly identified flow that is not part of the file transfer process would not exhibit these properties. The benefit of this additional heuristic is the confidence given to the operator in knowing the true incidence of false positives.

8.3 Summary

In this dissertation, I have demonstrated that it is feasible for ISPs to use full line-speed passive monitoring techniques for the accurate real-time identification of Grid bulk data traffic. These techniques are applicable both to clear text and encrypted control flows. The identification results can be used to enable traffic re-engineering and therefore improve network performance.

Appendix A

Real-Time Application Protocol Analyser Implementation

A.1 EPA Macros

The macro definitions behind the `EPA_READ` and `EPA_READ_AND_SKIP` methods introduced in Section 5.4.1 use either C `switch` statements or a GCC extension known as "computed `goto`." These two options are represented in Figure A.1. The base source code in the top panel shows a straightforward `while` loop that is conditional on there being enough data to work on. The actual calculations involved to determine whether enough data is present or not is represented by *HAVE_ENOUGH_DATA*—the precise details are not relevant here. The lower two panels show the cleaned-up results of the C preprocessor macro expansion.

In the "Switch" implementation, the function preamble contains a `switch` statement that covers the body of the function. The switch block is closed by the `EPA_END` macro. On first entry to the function, the `pt->lc` variable is 0, so execution continues normally. In order to yield, the `pt_yielded` flag is set to 0, causing the function to return with the constant `PT_THREAD_WAITING` after saving the resume-point (10, derived from the line number) back into `pt->lc`. When the function is re-entered, the switch statement jumps back to just before the point of return last time. Since the `pt_yielded` variable is now non-zero (due to the initialisation on line 4), execution continues. It is up to the calling function (a pseudo-scheduler) to take note of the return value and act accordingly.

The implementation for "Goto" is similar, but uses computed `goto` instead. Here, the address of an inline C label is taken. This avoids the disadvantage of the switch implementation whereby a nested switch statement cannot be used within the ProtoThread function, however compiler detection of potentially unsafe variable usage across resume points is not as reliable.

Although these constructs are somewhat complicated underneath, they greatly simplify the coding of the analyser routines.

Source

```
int Analyser::SomeFunction() {
  EPABuffer buf;

  EPA_BEGIN();

  while (!HAVE_ENOUGH_DATA) {
      EPA_YIELD();
  }

  // do something with buf.data

  EPA_END();
}
```

Switch

```
int Analyser::SomeFunction() {
  EPABuffer buf;

  char pt_yielded = 1;                        // EPA_BEGIN();
  switch (pt->lc) {
    case 0:

      while (!HAVE_ENOUGH_DATA) {
          pt_yielded = 0;
          pt->lc = 10;
          case 10:
            if (!pt_yielded)
               return PT_THREAD_WAITING;
      }

      // do something with buf.data
  }                                           // EPA_END();
  pt->lc = 0;
  return PT_THREAD_EXITED;
}
```

Goto

```
int Analyser::SomeFunction() {
    EPABuffer buf;

    char pt_yielded = 1;                      // EPA_BEGIN();
    if (pt->lc != NULL)
        goto *pt->lc;

    while (!HAVE_ENOUGH_DATA) {
        pt_yielded = 0;
        LC_LABEL10: pt->lc = &&LC_LABEL10;    // &&label is a compiler extension
        if (!pt_yielded)
          return PT_THREAD_WAITING;
    }

    // do something with buf.data

    pt->lc = NULL;                            // EPA_EXIT();
    return PT_THREAD_EXITED;
}
```

Figure A.1: The top listing, “Source”, represents the source code as entered by the programmer. The lower two listings, “Switch” and “Goto”, represent the code seen by the compiler in implementing the ProtoThreads.

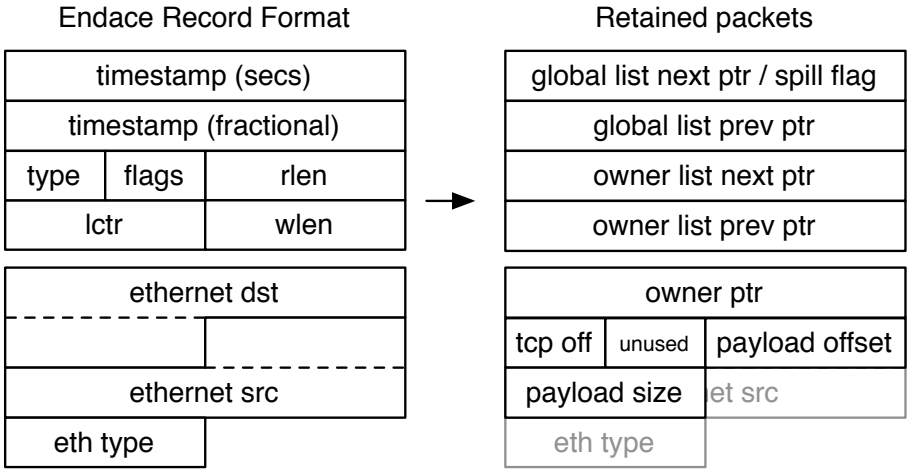


Figure A.2: DAG per-packet record header before and after a packet is converted into a retained packet and added to the global and per-flow linked lists. Each row in the diagrams above represent four byte quantities.

A.2 Header Fields

Figure A.2 shows the DAG per-packet record header before and after a packet is inserted into the retained packet queues. The fields are as follows:

global next ptr, global prev ptr Doubly linked global list.

spill flag Sentinel value stored in global next ptr field to indicate that this retained packet has been allocated on the heap.

owner next ptr, owner prev ptr Doubly linked list used by retained packet owner (a particular TCP flow or, later, a protocol analyser) to keep track of pending packets.

owner ptr Pointer to the owner of the packet so that it can decide whether to dispose or spill the packet if it is holding up the acknowledged pointer for the DAG card (see Figure 5.3).

tcp off Offset from start of packet to the TCP header. Not all retained packets have a TCP header, since when spilling a packet the TCP header may no longer be relevant.

payload off Offset from start of packet to the TCP payload.

payload size TCP payload length.

The pointer fields are each 32-bits in size. This assumes that the native pointer type on the host is 32-bits. A 64-bit address space could be accommodated by storing 32-bit offsets from the capture buffer for the linked list fields, and ensuring that the flow control blocks are allocated in a 32-bit memory window that can again be specified with an appropriate offset in the retained packet structure.

Glossary

DAG Hardware network monitoring card produced by Endace Measurement Systems.

DCS Department of Computing Science, at the University of Glasgow.

EGEE Enabling Grids for E-science in Europe

JANET The UKs education and research network.

LBL Lawrence Berkeley National Laboratory
Co-located with University of California at Berkeley

NCSA National Center for Supercomputing Applications

SDSC San Diego Supercomputer Center

UKERNA The United Kingdom Education and Research Networking Association.
The company that manages the JANET network.

Acronyms

AIMD Additive Increase Multiplicative Decrease
API Application Programming Interface
ARP Address Resolution Protocol
CPU Central Processing Unit
EDSM Event-Driven State Machine
ERF Endace Record Format
FSM Finite State Machine
FTP File Transfer Protocol
GCC GNU Compiler Collection
GPS Global Positioning System
GSI Grid Security Infrastructure
GSSAPI Generic Security Services Application Program Interface
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
IANA Internet Assigned Numbers Authority
IETF Internet Engineering Task Force
IP Internet Protocol
ISP Internet Service Provider
JANET Joint Academic NETwork
LAN Local Area Network
LBL Lawrence Berkeley Laboratory
MCAT Metadata Catalog
MSS Maximum Segment Size
MTU Maximum Transfer Unit
NAT Network Address Translation
NIDS Network Intrusion Detection System
NTP Network Time Protocol
PCI Peripheral Component Interconnect
RAID Redundant Array of Independent (or Inexpensive) Disks
RAM Random Access Memory
RFC Request For Comments
RPC Remote Procedure Call
RTT Round Trip Time

SLA Service Level Agreement
SMTP Simple Mail Transfer Protocol
SRB Storage Resource Broker
SSL Secure Sockets Layer
TCP Transmission Control Protocol
UDP User Datagram Protocol
VPN Virtual Private Network

Bibliography

[1] About the UK e-Science Programme. Available online at `http://www.rcuk.ac.uk/escience/`.

[2] Enabling Grids for E-science. Available online at `http://public.eu-egee.org/`.

[3] IEEE 802.3-2002 IEEE Standard for Information technology-Telecommunications and information exchange between systems–Local and metropolitan area networks-Specific requirements-Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications.

[4] Internet2 NetFlow Weekly Reports. Available online at `http://netflow.internet2.edu/weekly/`.

[5] Teragrid. Available online at `http://www.teragrid.org/`.

[6] D. Agarwal, J. M. Gonzalez, G. Jin, and B. Tierney. An Infrastructure for Passive Network Monitoring of Application Data Streams. In *Proceedings of Passive and Active Measurement Workshop*, 2003.

[7] J. L. Alberi, T. Chen, S. Khurana, A. Mcintosh, M. Pucci, and R. Vaidyanathan. Using Real-Time Measurements in Support of Real-Time Network Management. In *Proceedings of Passive and Active Measurement 2001*, 2001.

[8] 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)*, pages 161–163, 2000.

[9] N. Anerousis, R. Caceres, N. Duffield, A. Feldmann, A. Greenberg, C. Kalmanek, P. Mishra, K. Ramakrishnan, and J. Rexford. Using the AT&T labs PacketScope for internet measurements, design, and performance analysis. In *AT&T Services and Infrastructure Performance Symposium*, 1997.

[10] 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.

[11] J. Apisdorf, K. Claffy, K. Thompson, and R. Wilder. OC3Mon: flexible, af-

fordable, high performance statistics collection. In *Proceedings of INET 97*. National Laboratory for Applied Network Research, 1997. Available online at http://www.nlanr.net/NA/OC3mon/.

[12] D. Awduche, A. Chiu, A. Elwalid, I. Widjaja, and X. Xiao. Overview and Principles of Internet Traffic Engineering. RFC 3272 (Informational), May 2002. Available online at http://www.ietf.org/rfc/rfc3272.txt.

[13] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus. Requirements for Traffic Engineering Over MPLS. RFC 2702 (Informational), September 1999. Available online at http://www.ietf.org/rfc/rfc2702.txt.

[14] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC Storage Resource Broker. In *Proc. CASCON'98*, Toronto, Canada, 1998.

[15] S. A. Baset and H. Schulzrinne. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. Technical Report cucs-039-04, Department of Computer Science, Columbia University, 2004. Available online at http://www1.cs.columbia.edu/~library/TR-repository/reports/reports-2004/cucs-039-04.pdf.

[16] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May 1996. Available online at http://www.ietf.org/rfc/rfc1945.txt.

[17] R. Beverly. A Robust Classifier for Passive TCP/IP Fingerprinting. In *Passive and Active Network Measurement, 5th International Workshop, PAM 2004, Antibes Juan-les-Pins, France, April 19-20, 2004, Proceedings*, pages 158–167, 2004.

[18] G. D. Bissias, M. Liberatore, D. Jesnsen, and B. N. Levine. Privacy Vulnerabilities in Encrypted HTTP Streams. In *Proceedings of Privacy Enhancing Technologies Workshop (PET 2005)*, 2005.

[19] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Service. RFC 2475 (Informational), December 1998. Available online at http://www.ietf.org/rfc/rfc2475.txt.

[20] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. FFPF: Fairly Fast Packet Filters. In *Proceedings of OSDI'04*, 2004.

[21] H. Bos and K. Huang. Towards software-based signature detection for intrusion prevention on the network card. In *Proceedings of Eighth International Symposium on Recent Advances in Intrusion Detection (RAID2005)*, Seattle, WA, September 2005.

[22] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. RFC 2205 (Proposed Standard), September 1997. Available online at http://www.ietf.org/rfc/rfc2205.txt.

[23] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End Congestion Avoid-

ance on a Global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, October 1995.

[24] N. Brownlee. Traffic Flow Measurement: Experiences with NeTraMet. RFC 2123 (Informational), March 1997. Available online at `http://www.ietf.org/rfc/rfc2123.txt`.

[25] N. Brownlee. Traffic Flow Measurement: Meter MIB. RFC 2720 (Proposed Standard), October 1999. Available online at `http://www.ietf.org/rfc/rfc2720.txt`.

[26] N. Brownlee, C. Mills, and G. Ruth. Traffic Flow Measurement: Architecture. RFC 2722 (Informational), October 1999. Available online at `http://www.ietf.org/rfc/rfc2722.txt`.

[27] R. Buyya. Grid Computing Info Centre: Frequently Asked Questions (FAQ). `http://www.cs.mu.oz.au/~raj/GridInfoware/gridfaq.html`.

[28] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP). RFC 1157, May 1990. Available online at `http://www.ietf.org/rfc/rfc1157.txt`.

[29] CCITT (Consultative Committee on International Telegraphy and Telephony). *Recommendation X.509: The Directory—Authentication Framework*, 1988.

[30] cflowd: Traffic Flow Analysis Tool. Available online at `http://www.caida.org/tools/measurement/cflowd/`.

[31] J. Chase, A. Gallatin, and K. Yocum. End System Optimizations for High-Speed TCP. *IEEE Communications Magazine*, 39(4):68–74, 2001.

[32] Cisco Systems. NetFlow whitepaper, 2000. Available online at `http://www.cisco.com/warp/public/732/netflow`.

[33] J. Cleary, S. Donnelly, I. Graham, A. McGregor, and M. Pearson. Design Principles for Accurate Passive Measurement. In *Proceedings of PAM2000: The First Passive and Active Measurement Workshop*, 2000.

[34] G. Combs. The Ethereal Network Analyzer. Available online at `http://www.ethereal.com/`.

[35] J. Coppens, S. V. den Berghe, H. Bos, E. Markatos, F. D. Turck, A. Öslebö, and S. Ubik. SCAMPI: A Scalable and Programmable Architecture for Monitoring Gigabit Networks. In *Proceedings of E2EMON Workshop*, Belfast, UK, September 2003.

[36] R. L. Cottrell, A. Antony, C. Logg, and J. Navratil. iGrid2002 demonstration: Bandwidth from the low lands. *Future Gener. Comput. Syst.*, 19:825–837, 2003.

[37] R. L. Cottrell, C. Logg, and I.-H. Mei. Experiences and Results from a New High-Performance Network and Application Monitoring Toolkit. In *Proceedings of Passive and Active Measurement 2003*, 2003.

[38] L. Deri. Improving Passive Packet Capture: Beyond Device Polling. Available online at http://www.ntop.org/PF_RING.html.

[39] A. Dunkels, O. Schmidt, and T. Voigt. Using Protothreads for Sensor Node Programming. In *Proceedings of the REALWSN'05 Workshop on Real-World Wireless Sensor Networks*, Stockholm, Sweden, June 2005.

[40] Endace Measurement Systems. Endace DAG Coprocessor and Applications. Available online at http://www.endace.com/Library/Endace_DagCoprocessor_Rev_A.pdf.

[41] G. Farrache. bbFTP - Large files transfer protocol. Available online at http://doc.in2p3.fr/bbftp/.

[42] A. Feldmann. BLT: Bi-Layer Tracing of HTTP and TCP/IP. *WWW9 / Computer Networks*, 33(1-6):321–335, 2000.

[43] W. Feng, M. Gardner, and J. Hay. The MAGNeT Toolkit: Design, Evaluation, and Implementation. *Journal of Supercomputing*, 23(1):67–79, August 2002.

[44] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Available online at http://www.ietf.org/rfc/rfc2616.txt.

[45] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649 (Experimental), December 2003. Available online at http://www.ietf.org/rfc/rfc3649.txt.

[46] F. Georgatos, F. Gruber, D. Karrenberg, M. Santcroos, H. Uijterwaal, and R. Wilhelm. Providing Active Measurements as a Regular Service for ISP's. In *Proceedings of Passive and Active Measurement Workshop*, 2001.

[47] J. M. González and V. Paxson. pktd: A Packet Capture and Injection Daemon. In *Proceedings of Passive and Active Measurement Workshop*, 2003.

[48] V. Gough. RLog - a C++ logging library. Available online at http://pobox.com/~vgough/rlog.

[49] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload. In *Proceedings of SOSP'03*, 2003.

[50] J. Hall, A. Moore, I. Pratt, and I. Leslie. Multi-protocol visualization: a tool demonstration. In *MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 13–22. ACM Press, 2003.

[51] A. Hintz. Fingerprinting websites using traffic analysis. In *Proceedings of Privacy Enhancing Technologies workshop (PET 2002)*, 2002.

[52] M. Horowitz and S. Lunt. FTP Security Extensions. RFC 2228 (Proposed Standard), 1997. Available online at http://www.ietf.org/rfc/rfc2228.txt.

[53] A. Hussain, J. Heidemann, and C. Papadopoulos. Distinguishing between Single and Multi-source Attacks using Signal Processing. *Computer Networks*, 46(4):479–503, November 2004.

[54] IEPM web site. `http://www-iepm.slac.stanford.edu/pinger/`.

[55] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323 (Proposed Standard), May 1992. Available online at `http://www.ietf.org/rfc/rfc1323.txt`.

[56] C. Jin, D. Wei, and S. Low. FAST TCP: Motivation, Architecture, Algorithms, Performance. In *Proceedings of IEEE INFOCOM*, March 2004.

[57] S. Kalidindi and M. J. Zekauskas. Surveyor: An Infrastructure for Internet Performance Measurements. In *Proceedings of INET'99*, 1999.

[58] T. Karagiannis, A. Broido, N. Brownlee, K. Claffy, and M. Faloutsos. File-sharing in the Internet: A characterization of P2P traffic in the backbone. Technical report, UCR, November 2003.

[59] T. Karagiannis, A. Broido, M. Faloutsos, and K. Claffy. Transport layer identification of P2P traffic. In *Proc. ACM SIGCOMM conference on Internet measurement*, 2004.

[60] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: Multilevel Traffic Classification in the Dark. In *ACM SIGCOMM*, 2005.

[61] T. Kelly. Scalable TCP: Improving Performance in Highspeed Wide Area Networks. *Computer Communication Review*, 32(2), April 2003.

[62] K. Keys, D. Moore, R. Koga, E. Lagache, M. Tesch, and k. claffy. The Architecture of CoralReef: an Internet Traffic Monitoring Software Suite. In *PAM2001 – A workshop on Passive and Active Measurements*, 2001.

[63] C. Kreibich. NetDuDe (NETwork DUmp data Displayer and Editor). Available online at `http://netdude.sourceforge.net/`.

[64] Lawrence Berkeley National Laboratory Research Group. tcpdump. Available online at `http://www.tcpdump.org`.

[65] Lawrence Berkeley National Labs Network Research Group. libpcap. Available online at `http://ftp.ee.lbl.gov`.

[66] Z. Li, H. Zhang, Y. You, and T. He. Linuxflow: A High Speed Backbone Measurement Facility. In *Proceedings of Passive and Active Measurement Workshop*, 2003.

[67] C. Lowth. ROPE - IpTables Scripting Language. Available online at `http://www.lowth.com/rope/`.

[68] G. R. Malan and F. Jahanian. An extensible probe architecture for network protocol performance measurement. In *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 215–227, New York, NY, USA, 1998. ACM Press.

[69] R. Mann, R. Williams, M. Atkinson, K. Brodie, A. Storkey, and C. Williams. Scientific Data Mining, Integration and Visualisation. Technical Report UKeS-2002-06, National e-Science Centre, Nov 2002.

[70] W. Matthews and L. Cottrell. The PingER Project: Active Internet Performance Monitoring for the HENP Community. *IEEE Communications*, May 2000. Available online at `http://www.comsoc.org/ci/private/2000/may/Cottrell.html`.

[71] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of Winter USENIX Technical Conference*, Jan. 1993.

[72] A. McGregor, M. Hall, P. Lorier, and J. Brunskill. Flow Clustering Using Machine Learning Techniques. In *Passive and Active Network Measurement, 5th International Workshop, PAM 2004, Antibes Juan-les-Pins, France, April 19-20, 2004, Proceedings*, pages 205–214, 2004.

[73] A. J. McGregor, H. Braun, and A. Brown. The NLANR Network Analysis Infrastructure. *IEEE Communications*, May 2000.

[74] T. McGregor and H.-W. Braun. Balancing Cost and Utility in Active Monitoring: The AMP Example. In *Proceedings of INET2000*, 2000.

[75] D. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. RFC 1305 (Draft Standard), March 1992. Available online at `http://www.ietf.org/rfc/rfc1305.txt`.

[76] A. Moore, J. Hall, C. Kreibich, E. Harris, and I. Pratt. Architecture of a Network Monitor. In *Proceedings of Passive and Active Measurement Workshop*, 2003.

[77] A. W. Moore and K. Papagiannaki. Toward the Accurate Identification of Network Applications. In *Proceedings of Passive and Active Measurement Workshop*, 2005.

[78] A. W. Moore and D. Zuev. Internet traffic classification using bayesian analysis techniques. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 50–60, New York, NY, USA, 2005. ACM Press.

[79] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, January 1984. Available online at `http://www.ietf.org/rfc/rfc896.txt`.

[80] S. Ostermann. tcptrace. Available online at `http://irg.cs.ohiou.edu/software/tcptrace/index.html`.

[81] J. Paisley and J. Sventek. Real-time Detection of Grid Bulk Transfer Traffic. In *Proceedings of NOMS 2006 (to appear)*, 2006.

[82] R. Pang and V. Paxson. A High-Level Programming Environment for Packet Trace Anonymization and Transformation. In *Proceedings of the ACM SIGCOMM Conference*, August 2003.

[83] K. Papagiannaki, N. Taft, S. Bhattacharyya, P. Thiran, K. Salamatian, and

C. Diot. On the Feasibility of Identifying Elephants in Internet Backbone Traffic. Sprint ATL Research Report RR01-ATL-110918, Sprint ATL, 2001.

[84] A. Pásztor and D. Veitch. A Precision Infrastructure for Active Probing. In *Proceedings of Passive and Active Measurement Workshop*, 2001.

[85] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31:2435–2463, 1999.

[86] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis. An Architecture for Large-Scale Internet Measurement. *IEEE Communications*, 36(8):48–54, August 1998.

[87] M. Polychronakis and E. Markatos. Design of an Application Programming Interface for IP Network Monitoring. In *Proceedings of NOMS'04*, 2004.

[88] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Available online at `http://www.ietf.org/rfc/rfc793.txt`.

[89] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Standard), Oct. 1985. Available online at `http://www.ietf.org/rfc/rfc959.txt`.

[90] E. Rescorla. HTTP Over TLS. RFC 2818 (Informational), May 2000. Available online at `http://www.ietf.org/rfc/rfc2818.txt`.

[91] RIPE NCC Test Traffic Measurements. `http://www.ripe.net/ttm/`.

[92] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *USENIX 13th Systems Administration Conference - LISA '99*, 1999. Available online at `http://www.snort.org/`.

[93] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031, 2001. Available online at `http://rtg.ietf.org/rfc/rfc3031.txt`.

[94] D. Schuehler and J. Lockwood. TCP-Splitter: A TCP/IP Flow Monitor in Reconfigurable Hardware. In *Proceeding of Hot Interconnects 10 (HotI-10)*, 2002.

[95] SDSC. SRB Projects. Available online at `http://www.sdsc.edu/srb/index.php/Projects`.

[96] 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.

[97] S. Sen and J. Wong. Analyzing peer-to-peer traffic across large networks. In *Second Annual ACM Internet Measurement Workshop*, 2002.

[98] R. Shorten and D. Leith. H-TCP: TCP for high-speed and long-distance networks. In *Proc. PFLDnet*, Geneva, CH, February 2003.

[99] D. X. Song, D. Wagner, and X. Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *USENIX Security '01*, 2001.

[100] A. Thurston. Ragel State Machine Compiler. Available online at `http://www.elude.ca/ragel/`.

[101] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf—The TCP/UDP Bandwidth Measurement Tool. http://dast.nlanr.net/Projects/Iperf/.

[102] United Kingdom Education and Research Networking Association. SuperJANET5: An Architecture for Diversity. Available online at http://www.ja.net/sj5/requirementsanalysis/an-architecture-for-diversity.pdf.

[103] S. Waldbusser. Remote Network Monitoring Management Information Base. RFC 2819 (Standard), May 2000. Available online at http://www.ietf.org/rfc/rfc2819.txt.

[104] D. Watson, G. R. Malan, and F. Jahanian. *An extensible probe architecture for network protocol performance measurement*, volume 34, chapter 1, pages 47–67. John Wiley & Sons, 2004.

[105] E. Weigle and W. Feng. TICKETing High-Speed Traffic with Commodity Hardware and Software. In *Proceedings of Passive and Active Measurement Workshop*, 2002.

[106] P. Wood. libpcap-mmap. Available online at http://public.lanl.gov/cpw/.

[107] S. Zander, T. Nguyen, and G. Armitage. Automated Traffic Classification and Application Identification using Machine Learning. In *Proceedings of the IEEE Conference on Local Computer Networks 30th Annversary (LCN'05)*, 2005.

[108] S. Zander, T. Nguyen, and G. J. Armitage. Self-Learning IP Traffic Classification Based on Statistical Flow Characteristics. In *Passive and Active Network Measurement, 6th International Workshop, PAM 2005, Boston, MA, USA, March 31 - April 1, 2005, Proceedings*, pages 325–328, 2005.