mmdump: A Tool for Monitoring Internet Multimedia Traffic

Jacobus van der Merwe[*], Ramón Cáceres[†], Yang-hua Chu[‡], Cormac Sreenan[§]
kobus@research.att.com, ramon@vindigo.com, yhchu@cs.cmu.edu, cjs@cs.ucc.ie

Abstract

Internet multimedia traffic is increasing as applications like streaming media and packet telephony grow in popularity. It is important to monitor the volume and characteristics of this traffic, particularly because its behavior in the face of network congestion differs from that of the currently dominant TCP traffic. To monitor traffic on a high-speed link for extended periods, it is not practical to blindly capture all packets that traverse the link. We present *mmdump*, a tool that parses messages from RTSP, H.323 and similar multimedia session control protocols to set up and tear down packet filters as needed to gather traces of multimedia sessions. Unlike tcpdump, dynamic packet filters are necessary because these protocols dynamically negotiate TCP and UDP port numbers to carry the media content. Our tool captures only packets of interest for optional storage and further analysis, thus greatly reducing resource requirements. This paper presents the design and implementation of *mmdump* and demonstrates its utility in monitoring live RTSP and H.323 traffic on a commercial IP network. The preliminary results obtained from these measurements are presented.

1 Introduction

Recent years have seen increasing use of the Internet to send and receive audio and video, including streaming playback of music and news, as well as real-time voice telephony and conferencing. This traffic is expected to continue growing, driven by improvements in PC performance, residential access bandwidth, and media coding algorithms. Whilst the trends and behavior of Web traffic have been studied extensively, multimedia traffic has yet to be studied in detail. Multimedia applications typically use UDP transport, demand relatively large and constant data rates, and react slowly, if at all, to network congestion. As this traffic grows, its impact on network performance may be significant. It is important for network designers to understand the nature of multimedia traffic.

Internet traffic measurements are commonly performed using the *tcpdump* utility, which can be used to monitor packets for a particular application-level protocol by filtering based on the appropriate TCP/UDP port number. Use of *tcpdump* for multimedia traffic is complicated because the majority of multimedia applications use dynamically assigned UDP port numbers. For example, protocols such as the Real Time Streaming Protocol (RTSP) [19], the Session Initiation Protocol (SIP) [6], and H.323 [7] use a well known TCP port number to initiate a multimedia session. Once the session is established, the protocols negotiate other TCP or UDP port numbers dynamically for media control traffic and media data traffic. To address this problem we have created a new utility we call *mmdump* that is based on *tcpdump* but makes use of protocol-specific *parsing modules* to determine the dynamic set of ports that need to be monitored.

In this paper we present the design and implementation of *mmdump*. *mmdump* contains a parsing module for each multimedia control protocol. All traffic received on the well known control port is passed to the parsing module in question. The parsing module identifies individual control sessions in this aggregate control stream, and parses the control messages to extract the dynamically assigned port numbers. The parsing module then dynamically changes the packet filter to allow packets associated with these ports to be captured. Architecturally, *mmdump* departs from *tcpdump* by maintaining state for each multimedia session. This is necessary because *mmdump* needs to associate arriving packets with individual sessions and later report statistics of the sessions. The situation is made worse by issues such as packet loss and asymmetric routing. We present our approach to these problems in the current implementation of *mmdump* and suggest improved approaches.

We also present results obtained using *mmdump* to monitor multimedia traffic in AT&T's commercial IP network. The version of *mmdump* used included RTSP and H.323 parsing modules; we have since developed a rudimentary SIP parser. The varied types of analysis that we present for traffic from different multimedia control protocols highlight the versatility of *mmdump*.

The rest of this paper is organized as follows. Section 2 provides background on *tcpdump* as well as RTSP and H.323. Section 3 explains the structure and operation of *mmdump*. Section 4 presents results demonstrating the use of *mmdump* on live multimedia traffic. Section 5 summarizes related work, and Section 6 concludes the paper.

[*] AT&T Labs–Research, Florham Park, NJ, USA
[†] Vindigo, New York, NY, USA
[‡] Carnegie Mellon University, Pittsburgh, PA, USA
[§] University College Cork, Cork, Ireland

2 Background Information

Given that *mmdump* is based on and extends *tcpdump*, we give a brief overview of *tcpdump* in this section. In addition, we briefly discuss two example multimedia control protocols that are used to negotiate port numbers for streaming content.

2.1 Structure of *tcpdump*

The *tcpdump* utility provides a popular mechanism for monitoring packet transmissions. *tcpdump* builds on top of the *libpcap* library, which provides two key functions: an abstraction for dealing with different types of network interfaces, and the ability to compile a filter expression for use by a packet filter. The library provides a common interface to different ways of performing packet filtering. For example, on a system with the BSD Packet Filter (BPF) [11], filtering is done in kernel space and *libpcap* simply passes the compiled filter expression to the kernel. *libpcap* can also perform the packet filtering itself (in user space) when required. This is used on systems where the kernel does not support packet filtering, and when *tcpdump* is reading packets from a previously generated raw dump file, rather than directly from the network.

In normal operation, *tcpdump* is run with a command-line expression indicating all packets of interest. The grammar and syntax used for this expression is fairly high level. For example, the expression `host 135.207.26.201 and tcp port 554` indicates an interest in all TCP packets using port 554 that are either originating from or going to the host with the specified IP address. This expression is passed to the *libpcap* library at startup where it is compiled into an intermediary tree structure. The tree is then optimized and the resulting tree is translated into a contiguous filter expression which is installed in the operational packet filter.

For *tcpdump*, all packets that pass through the installed filter will either be logged to file, or be passed to a printing module in the *tcpdump* part of the code. In the latter case, print functions for successively higher layers of the protocol stack typically print out parts of the packet. For example, for a UDP packet carrying a Sun RPC request, the *print-ethernet* function will call *print-ip*, which in turn will call *print-udp* and then *print-sunrpc*. For a more detailed discussion of *tcpdump* please refer to [9, 8, 2].

While extremely popular and successful as a monitoring tool, *tcpdump* is unable to efficiently monitor multimedia traffic, since the majority of multimedia applications use dynamically assigned UDP or TCP port numbers for media transfer. This is the case with popular multimedia protocols such as RTSP and H.323, in which a control interaction using a well known port is used to negotiate the set of dynamic port numbers to be used for media transfers. By their nature, dynamically assigned port numbers cannot be specified from the command line, meaning that *tcpdump* in its current form cannot be used to monitor this type of traffic. If a given multimedia protocol normally picks a port from a small range of port numbers, it is of course possible to statically specify the whole range from the command line and perform post-processing to extract the data of interest. As we will discuss in Section 4.1, using *tcpdump* in this manner is very inefficient in terms of disk space and does not scale to processing packets on fast network links.

2.2 Real Time Streaming Protocol (RTSP)

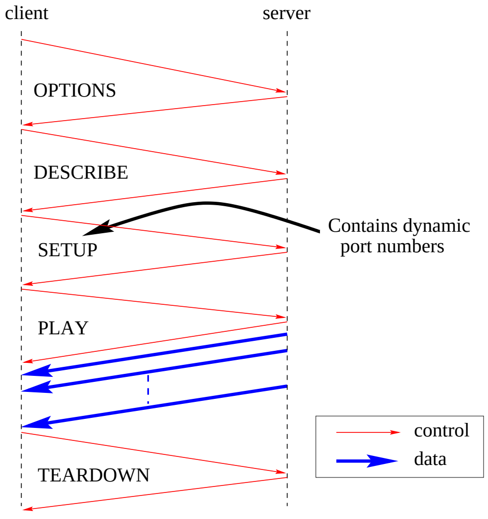

Figure 1: Dynamic port number assignment with RTSP

The Real Time Streaming Protocol (RTSP) is becoming the dominant control protocol for streaming content on the Internet. Figure 1 depicts a sample interaction between an RTSP client and server. RTSP is used to set up and control (pause, forward, etc.) the playback of streaming content across the Internet. RTSP is a classic request-response protocol, but also allows pipelining of messages to reduce latency. Protocol interaction starts with an OPTIONS request/response whereby the client and server establish mutual capabilities. The client then issues a DESCRIBE request for the media stream it is interested in. The response from the server contains media specific information about the stream, e.g. the encoding used, the clip length and the average bit rate. Depending on the particular session, more than one media stream might be described in a single DESCRIBE response message.

After DESCRIBE, the client issues a SETUP request which contains the set of protocols and port numbers (or range of port numbers) on which the client is willing to the receive the media stream. For RTSP this is normally UDP and a dynamically chosen port number, although it is also possible to use RTSP in “interleaved mode” where the data stream is interleaved on the original TCP control connection. This interleaving is typically only used to allow streaming through a firewall. The server selects one of these options and a port number and sends it back to the client in the response message.

Following these exchanges the client can issue a PLAY request to start the streaming and can issue PAUSE and other control requests for the stream. The session normally ends with a TEARDOWN request at which time the TCP connection is also terminated[1].

[1] This summary of the RTSP protocol reflects our monitoring of its usage in practice. It is compliant with the RTSP specification, but the specification allows several variations, for example the use of UDP as the transport mechanism for RTSP and the tearing down of the RTSP control connection without terminating the RTSP session.

2.3 H.323 conferencing control protocols

Conferencing and packet telephony represent another class of applications that make use of a separate control protocol to dynamically negotiate port numbers for media transfer. H.323 is a popular example of such a protocol. In principle H.323 operates in a manner similar to RTSP, but we describe the details here for the sake of completeness. Figure 2 depicts a sample H.323 exchange between caller and callee. Interaction starts with the caller sending a SETUP message on a well known TCP port to the callee. This exchange is on the first of two TCP connections which is called the Q.931 channel. The callee responds with an ALERTING message followed by a CONNECT message. The CONNECT message contains the port number for the second TCP connection between caller and callee which is called the H.245 or conference control channel. At this point the first TCP connection may be disconnected. Interaction on the H.245 channel starts with an exchange of messages to determine terminal capabilities and for determining the master and slave roles between the two terminals. The sender (of subsequent media) then sends an Open Logical Channel message to the receiver. In the Internet environment this message contains the RTCP port number on which the sender wants to receive RTCP reports about the quality of reception [18]. The receiver responds with an Ack message which contains the RTP port number on which the media stream should be received and an RTCP port number on which to receive RTCP sender reports. The Open Logical Channel message always originates from the sender of a data. As indicated in Figure 2 a two-way conversation will therefore require an Open Logical Channel and Ack pair of messages in both directions. The second TCP connection remains in place for the duration of the call and terminates after sending an End Session message.

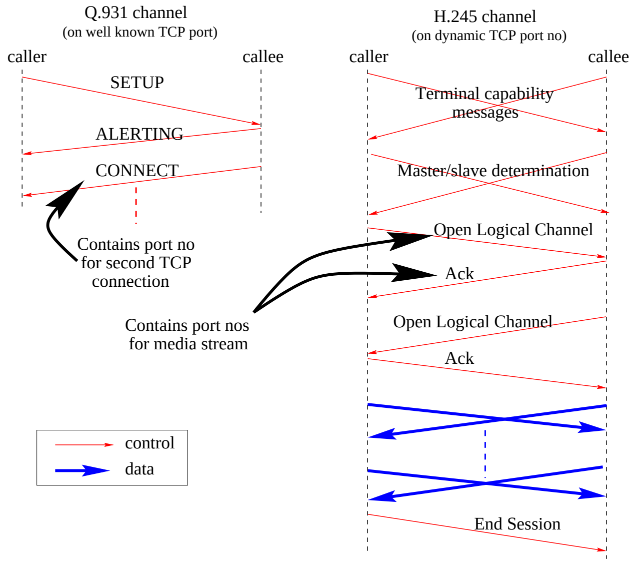

Figure 2: Dynamic port number assignment with H.323

3 Design, implementation and operation of *mmdump*

mmdump extends *tcpdump* by adding parsing modules for multimedia session control protocols and by allowing these parsers to dynamically change the packet filter to accept packets on the dynamically assigned port numbers. As per the normal functioning of *tcpdump*, packets that pass through the filter can be displayed by means of protocol-specific print modules, or can be logged to a file for post processing. This arrangement is depicted in Figure 3 and discussed in more detail in this section. We describe below our parsing modules for both RTSP and H.323. Note that all existing *tcpdump* functions remain in *mmdump* and the latter is therefore a superset of *tcpdump*. However, in order to explain the functionality of *mmdump* it is easier to refer to it as if it were a completely different tool.

When gathering lengthy traces on high-speed links, *mmdump* is commonly used in two stages. During the first stage, only the messages containing dynamically assigned port numbers are parsed, the packet filter is updated and all packets that pass through the filter are dumped into a file for later analysis (this includes all control packets and all data packets). In this mode of operation the parsing modules are only concerned with messages containing the dynamically assigned port numbers. Raw dump files generated in this manner can be post processed again using *mmdump*. In these cases the parsing module might extract information from other messages, e.g. in the case of RTSP the URL of media objects, the type of encoding used and the length of objects might be of interest. It is also possible to use *mmdump* in a one stage process whereby all information of interest is printed online and no packets are logged.

3.1 Structure of *mmdump*

The multimedia control protocols make use of well known port numbers. When started, *mmdump* sets up a default filter to capture all packets that belong to these control connections to bootstrap the monitoring process. This filter is set up to receive all packets for all connections traversing the probe point that *mmdump* is monitoring.

For each of the multimedia applications of interest a parsing module has to be supplied. All packets that arrive on a particular well-known port number are passed to the corresponding parsing module for processing. Figure 4 shows the functionality that each parsing module needs to supply.

mmdump maintains state for each active “session”. As shown in Figure 4, the first action required by a parsing module is to do a session lookup. A session is defined as a unique instance of a control protocol interaction, e.g. an RTSP client communicating with a RTSP server, or two H.323 peers communicating. A session lookup therefore involves a matching of source and destination addresses and port numbers in the received packet against the equivalent values in the stored session state. For H.323, the second TCP control connection (the H.245 connection), has to be associated with the first control connection (the Q.931 connection). In this case, the session lookup therefore has to match the incoming packet against both these control connections associated with the same session. While the basic session lookup is fairly generic, i.e. matching IP

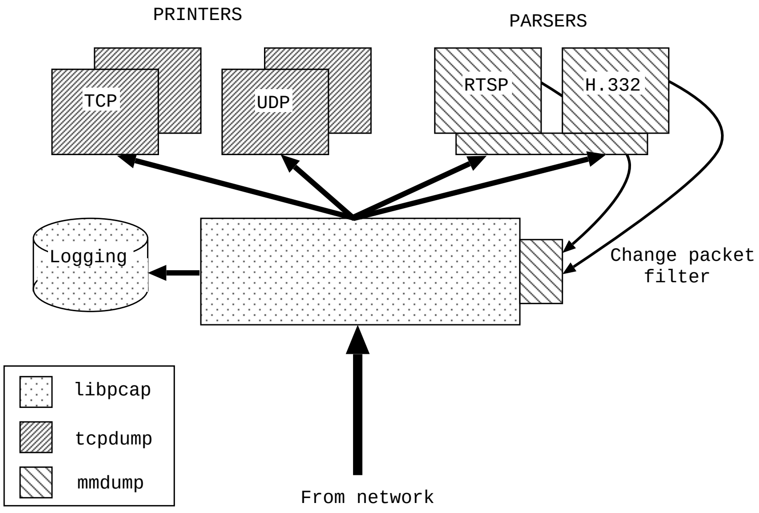

Figure 3: Architecture of *mmdump* in relation to *tcpdump*

addresses and port numbers, protocol-specific variations such as the aforementioned make it awkward to efficiently separate this functionality out in a generic way. If the session lookup was successful, the retrieved session state is used, or if it failed, a new session structure is allocated.

Maintaining state in the tool is a significant departure from the largely stateless operation of *tcpdump*. *tcpdump* does maintain some state, e.g. in order to print out relative rather than absolute sequence numbers for TCP packets. The state maintained by *mmdump* is different in that it keeps track of multimedia sessions, each of which can have more than one TCP or UDP connections. This means that *mmdump* keeps a lot more state than *tcpdump* and further needs to match different connections with the same multimedia session. As indicated above, new session state can be created when the first TCP packet for a particular session is received. Session state can be removed when the TCP FIN packet is received on the control connection for RTSP, and on the H.245 connection for H.323. Depending on the *mmdump* mode of operation, a summary of a session will typically be produced when session state is removed. Because of the size of the state that is maintained for each session, performing garbage collection of stale session state is essential in *mmdump*.

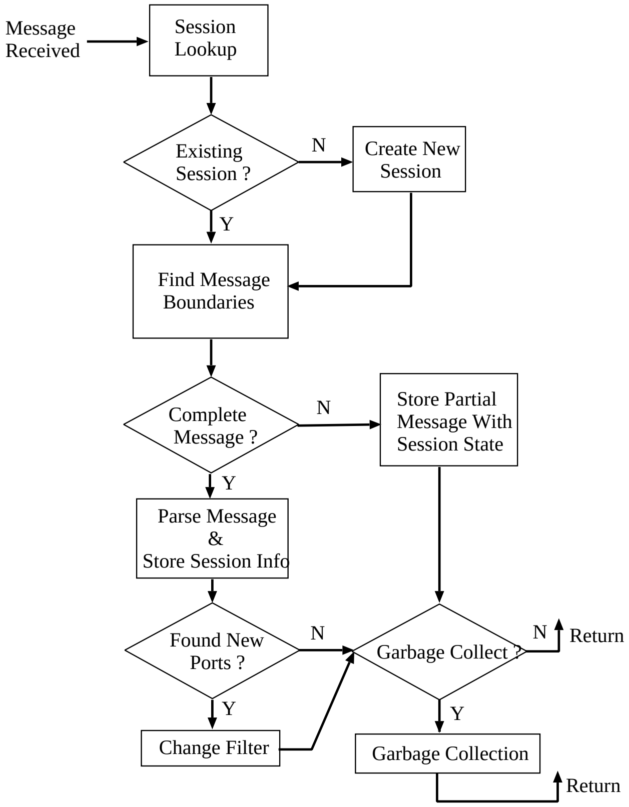

Figure 4: Functioning of a parsing module

Next, *mmdump* has to determine if a complete higher layer protocol message has been received. This function is by necessity protocol-specific. RTSP, which is a text-based protocol, requires a fairly complicated set of rules to determine when the end of a message is found. An H.323 control messages, on the other hand, is encapsulated in a lower level frame which has a message-length field. If a complete control protocol message has been received, the message is passed to a parsing engine, if not, the packet is stored in a per-session buffer to be used together with subsequent packets received for this session. The current implementation of this per-packet buffer does not take TCP sequence numbers into account and simply treats packets in the order in which they were received. This is clearly problematic in an IP environment where both packet reordering and packet loss can happen. Especially for RTSP traffic where different RTSP messages often span several IP packets, or have fragments of different RTSP messages in the same IP packet. For H.323, control messages seem to be contained in one (or two) IP packets, with a single H.323 message per packet. Since both control protocols in question (and indeed others that are of interest) make use of TCP, it should be possible to extend the implementation with a generic TCP module, which could pass to the parsing modules only in-order TCP segments. We are currently investigating this possibility.

The protocol-specific parsing engine tries to parse the message passed to it, putting extracted information in a session structure supplied by the parsing module proper. This separation of functionality allows a different parsing engine to be used without the need to change any of the mmdump logic and functionality[2]. At the very least the parsing engine will try to determine any dynamically negotiated port numbers. Depending on the way the tool is used (i.e. one-step online, or two-step online and off-line), the parsing engine also extracts other information from the control protocol. Should the parsing engine fail to correctly parse a message that was believed to be intact, perhaps because of the simple reassembly described above, the message is discarded and an event count is incremented.

If the parsing engine was able to extract any negotiated port numbers, this fact is relayed back to the main parsing function by means of a flag in the shared session structure. The parsing module can now invoke new functions exported by the *libpcap* library to dynamically change the filter expression so that packets associated with this port number can also pass through the packet filter.

The interface between the parsers and the packet-filter level is very simple and consists of two function calls: `change-filter()` and `do-filter()`. `change-filter()` allows a parser to either add or delete a port number for a particular address and protocol type to the filter expression. Alternatively a parser can request that all ports associated with a particular address be deleted. Calling `change-filter()` does not result in the immediate update of the real packet filter, rather the requested change is noted at the packet-filter level, and when a parsing module calls `do-filter()`, the actual filter change takes place. This allows the parsing module to make a number of related changes to the packet filter in one go, for example to add both RTP and RTCP port numbers to a specific address. This is desirable, because as explained below, the actual generation of a new packet filter is currently an expensive operation which should not be performed unnecessarily.

As explained in Section 2.1, for regular *tcpdump* a command-line filter expression is compiled once (using the function `pcap-compile()`), into an intermediate tree structure which is then optimized to produce a contiguous filter expression in a form which can be installed in a packet-filter state machine. In our initial proof-of-concept implementation, we made use of this same interface by producing a long ASCII filter expression for input to `pcap-compile()` every time that `do-filter()` was called. Generating the intermediate tree structure is however a very expensive operation and this approach was therefore very inefficient.

In our current implementation we bypass the standard compilation process by exploiting the fact that the filter expressions that we generate always follow a simple pattern. In particular, a command-line version of the filter expression used by *mmdump* will always be of the following nature: `tcp port X or (host A and port A1 or host A and port A2) or (host B and port B1) etc`. We therefore generate the intermediate tree structure directly by simply walking through the list of current entries in the filter table and AND'ing or OR'ing the building blocks of the tree structure together as needed. As before this intermediate tree structure is then optimized (through a new `simple-pcap-compile()` function) and turned into a contiguous filter expression for the actual packet filter.

While much more efficient than our initial attempt, the optimization process still needs to be run for the complete filter expression every time a parsing module calls `do-filter()`. A better solution, keeping the intermediate tree structure that can be modified based on instructions from the parsers, is not possible with the current *libpcap* implementation, as the intermediate tree structure is "consumed" in the optimization process. We understand that a new version of the *tcpdump* family of tools [2] is being written and that the needs of *mmdump* (and indeed other measurement work) for dynamic and incremental filters will be incorporated in the *libpcap* library.

Returning to Figure 4, the final function that a parsing module may have to perform is garbage collection. As described above, session state is normally removed when a TCP FIN message is received for the control connection. However, because of effects such as packet losses or route changes, the probe point might never receive the FIN packet and garbage collection has to be performed to remove stale session state. In our current implementation, garbage collection can be performed when triggered by some "scarcity" of resources, such as the number of sessions reaching a certain threshold or based on a timeout. Similarly, in the absence of more accurate information, sessions are deemed stale when their duration exceeds a certain threshold, or when they have not seen any activity on all the streaming ports for a certain period of time. The latter approach, while being more accurate, adds considerable overhead as it means that a session lookup has to be performed for every (or every nth) data packet.

3.2 Using *mmdump*

Selection of a particular multimedia protocol to monitor is by a command-line option: `-R n` for RTSP and `-H n` for H.323, where `n` is a small number controlling the amount of online processing and the verbosity of the output that *mmdump* produces. For example, `-H 0`, will do the minimal amount of online extraction of information and is often used in conjunction with the raw-write *tcpdump* option (`-w <filename>`), which saves the packets to disk for later processing when *mmdump* is used in two stages. `-H 1` causes *mmdump* to perform online extraction of protocol specific information and can be used either online or off-line, the latter typically with the *tcpdump* raw-read option (`-r <filename>`). With `n>0`, *mmdump* produces session specific records: For RTSP each session record shows the session related information, such as the start and end time, and the client and server addresses. In addition media-

[2] For example, a new H.323 library, capable of parsing the FastConnect option in the latest version of the specification was recently added to *mmdump* for monitoring voice traffic in a voice over IP trial.

specific information (e.g. format, size) for each media element (e.g. an audio clip, a background image) is shown including the URL and the clip length of each element. For H.323, each session record contains the IP addresses and phone numbers of the participants, the call duration and information about the audio codec and H.323 vendor whose software was used.

Normal *tcpdump* operation includes the notion of a "snaplen" (snapshop length), the maximum number of bytes from each packet that will be captured. *tcpdump* allows snaplen to be specified from the command-line or uses a default snaplen if none is specified. With *mmdump*, we need to capture the complete control messages in order to parse them correctly. The snaplen should therefore be set to the MTU (Maximum Transmission Unit) on a particular medium. In general, however, there is no need to capture the complete data packets. We have therefore added an option (`-D`), again used in conjunction with the `-w` option, to reduce the snaplen applied to data packets to include only header information of such packets. This dramatically reduces the storage requirements when raw dump files are used.

4 Results

In this section we present preliminary results obtained from our use of the *mmdump* tool. In Section 4.1 we present results of using *mmdump* to monitor RTSP traffic: Section 4.1.1 contains results of a single RTSP presentation in a controlled environment, while Section 4.1.2 presents measurement results from a probe point in AT&T's commercial IP network. In Section 4.2 we present a similar set of results for the use of *mmdump* on H.323 traffic: Section 4.2.1 is for a single H.323 session in a controlled environment and Section 4.2.2 presents results for H.323 traffic from the same probe point in the AT&T network.

The results presented are meant to show some of the possibilities of the tool rather than conclusive results about the use of streaming media in the Internet.

4.1 RTSP Results

4.1.1 Individual session in controlled environment

In this case a single RTSP presentation was viewed by means of a RealPlayer [15] client from a PC running Microsoft Windows. A Linux PC on the same Ethernet segment was running the *mmdump* tool to capture the interaction. The presentation in question was CNN Headline News [4], which was streamed from the Internet.

The CNN Headline news presentation consists of a small video section in the top left corner of the display area. Below the video section is a text window for presenting the latest news in text format (this normally contains a link to the CNN web site), in addition to an advertising section and a hyperlink to provide feedback. The right-hand side of display area consist of hyperlinks to other news-related streaming presentations.

While not visible to the user the presentation in question is served from two separate servers in different domains. This requires two RTSP sessions, the details of which are presented in Tables 1 and 2 and in Figure 5.

Table 1 shows the "base-URL" served by each server as well as the number of RTSP control packets going between the client and each server. Note that because of the location of the *mmdump* machine relative to the client machine, this traffic trace contained packets going in both directions between the client and server machines. Because of asymmetric routing in IP networks this is not the case in general.

Table 2 shows the URL extension, which together with the base-URL presents the complete URL for each object that is part of the presentation. Also shown in the table is the UDP port number chosen by the client, the number of UDP packets used to stream each object to the client, as well as a file-type field.

Figure 5 shows the packet arrival information for all UDP streams on a common timeline. The offset on the y-axis is used to depict the port number used for streaming the media. Each small vertical line on a horizontal line indicates a packet arrival event. Figure 5 clearly shows how the first object "streamed" to the client is a SMIL file [20], index.smi. This object in fact contains a description of the presentation which includes the layout of the presentation display, the various objects associated with each region of of the display, the location of each such object and optionally a timeline indicating when different objects should be be displayed. It is thus from the SMIL file that the client learns that some of its media should be retrieved from a different server. As indicated in Table 2 the actual "interesting" media content is streamed from the cnn.com domain, while several "support objects" like the background and links to other SMIL files are streamed from the real.com domain.

With reference to Figure 5, the presentation starts with a short signature tune (sting.rm) with an accompanying short animation (1.swf). Next the background image (back.jpg), the static advertisement (ad.gif) and text supporting text (left.rt) is streamed and presented. This is followed by an audio/video advertisement clip (ad1-28.rm) and text to make up the hyper-links in the presentation (links.rt). In the mean time more text is streamed (nowplaying_headlinenews.rt) and finally the audio and video clip for the actual headline news is streamed (headlines.rm28.rm).

4.1.2 Sessions in the public Internet

We gathered packet traces from a measurement probe inside the public Internet, more specifically from a T3 private peering link inside AT&T's commercial IP network. This placement of the probe machine means that in general both directions of interaction between two hosts will not necessarily be visible at the probe point due to asymmetrical routing. In all cases the traces were anonymized as soon as they came off the link under study, before writing any packet headers to stable storage. We collected traces on a dedicated 500MHz Alpha workstation that ran Digital UNIX and was attached to the link under study.

The traces analyzed in this section were gathered from our probe point in New York City for the week 15 April 1999 to 22 April 1999. Since at the time *mmdump* was still being tested the traces were gathered with a regular *tcpdump* capturing all packets on TCP port 554 and all UDP packets in the port ranges from 6970 to 7040

Session No	Session base URL	TCP packets	
		Client to server	Server to client
0	albany-b.real.com/showcase/channels/cnn_headlines/gold/	66	75
1	realchannel.cnn.com/	65	47

Table 1: Two RTSP sessions associated with single CNN headline news presentation

Session No	Media Stream	Client Port	URL extension	UDP packets	UDP Bytes	File type
0	0	6970	index.smi	5	1656	SMIL
	1	6972	audio/sting.rm	26	13620	Real Audio/Video
	2	6974	flash/1.swf	18	5357	Shockwave Flash
	3	6976	pix/back.jpg	30	14532	JPEG
	4	6978	text/left.rt	8	2014	Real Text
	5	6980	pix/ad.gif	17	6796	GIF
	6	6982	text/links.rt	11	4032	Real Text
	7	6984	text/feedback.rt	5	472	Real Text
	8	6992	text/nowplaying_headlinenews.rt	4	384	Real Text
1	0	6986	ads/ad1_28.rm	81	23609	Real Audio/Video
	1	6994	channel/headlines.rm28.rm	3991	1578597	Real Audio/Video

Table 2: Eleven media streams associated with single CNN headline news presentation

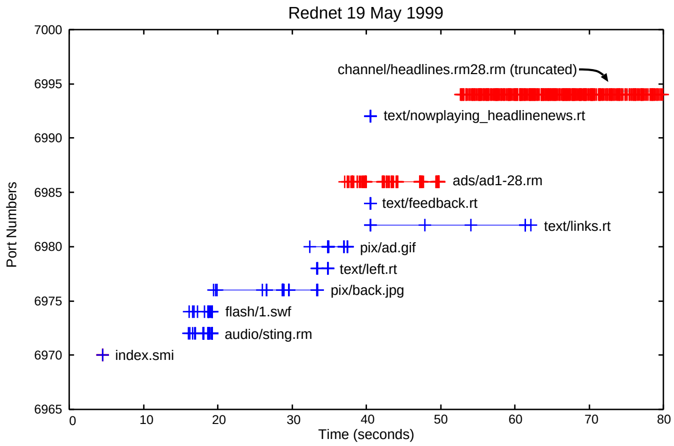

Figure 5: UDP activity for each media stream for CNN headline news from Real Networks (80 seconds shown)

inclusive, and *mmdump* was used exclusively in post processing mode. This proved a useful means to gather data to test *mmdump*, but also served to convince us about the need for a tool like *mmdump*. We captured the whole packet length for all TCP packets, as this is required for the RTSP protocol parsing, but only the first 136 bytes for UDP packets. The trace files for the week resulted in approximately 15 Gbytes worth of gzip'ed files. A new trace file was generated each 30 minutes and typically varied from below 10 Mbytes to well over 100 Mbytes depending on the time of day. Later using *mmdump* to trim these files to the traces it would have created from the original data resulted in a 60% to 80% reduction in required disk space per file.

Traffic Characteristics

One of the main questions we hope to address with this work is to determine the amount of streaming media relative to other Internet traffic and to monitor any changes in the longer term.

The issue of asymmetric routing has been mentioned a number of times in this paper. It turns out that for RTSP-related traffic a very small percentage of traffic at the probe point was in fact visible in both directions between client and server. This can potentially lead to erroneous conclusions about the relationship between control and data traffic for streaming media. For example, the network locality of a popular server might generate a lot of control traffic seen going from clients to servers, without a reciprocal contribution in data streamed from the server if that traffic does not pass the probe point. In Figure 6 we therefore show the control (RTSP/TCP) and data (UDP) traffic volumes (in number of packets) going only from servers to clients. This appears to be a reasonable comparison of the relationship between control and data. As before packet counts were generated for every half hour of the trace data.

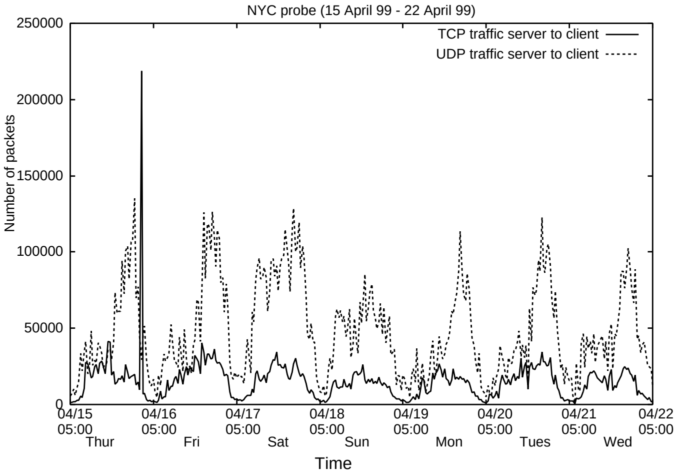

Figure 6: RTSP and related UDP packet counts

One observation regarding Figure 6 is that peak hours are drastically shifted towards the late evening hours. This contrasts with aggregate TCP traffic characteristics (not shown) which normally have very clear peaks during office hours. From Figure 6, activity over weekends are not significantly lower than over weekdays, only more evenly spread over all hours.

Figure 7 shows the packet length distribution for RTSP-related (i.e. streaming) UDP traffic. Significant peaks are at packet lengths much shorter than typical Maximum Transmission Unit (MTU) sizes. Some of these can probably be attributed to concerns about delay and latency for fairly low bitrate voice encoders and the distribution will in general be influenced by popular voice and video encoding and packetization schemes. Packet lengths for RTSP-related TCP traffic follow the familiar distribution with 40 bytes corresponding to TCP ACK, FIN, and SIN packets, and two MTU related peaks at 576 and 1500 bytes.

Content Analysis

In addition to looking at the traffic generated by streaming media in a general sense, *mmdump* allows us to look at a number of application- or protocol-specific issues. Figure 8 presents information about the URLs extracted from our week-long trace. Only 3074 unique URLs were observed in the trace. Only domain names combined with the requested object names were taken into account in determining uniqueness. I.e. the same object being served from two different machines in the same domain would not be considered unique. Figure 8 shows the number of references to each of the most popular 1000 objects. This Zipf-like distribution, showing that relatively few objects are extremely popular, has strong implications for caching strategies for multimedia objects.

Rate Adaptation

As a final example of the capabilities of *mmdump*, we have investigated the transmission rate of a single media stream and considered its interaction with the application control protocol. The RTSP protocol has a SET_PARAMETER method that can be used to set arbi-

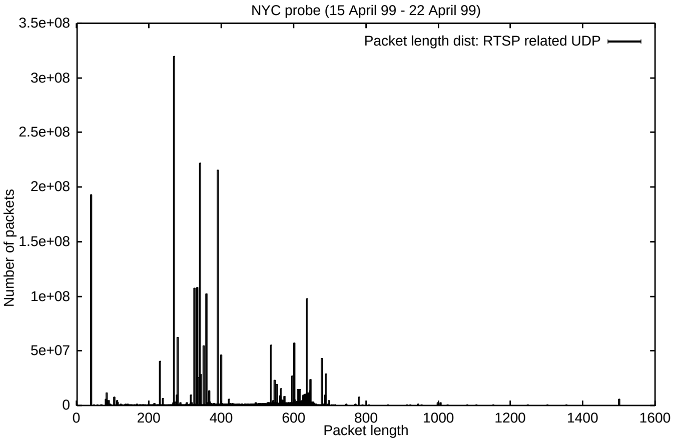

Figure 7: UDP packet length distribution for RTSP related traffic

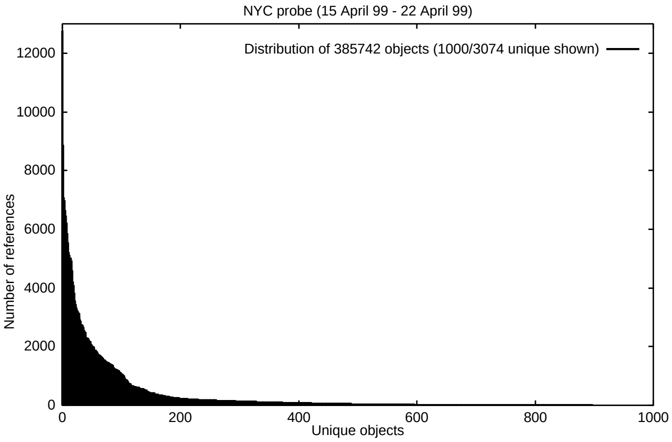

Figure 8: Distribution of URLs

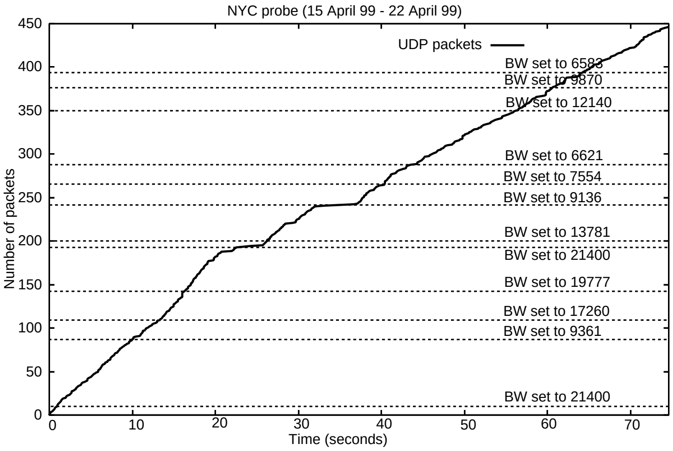

Figure 9: Packet arrivals for a single live UDP stream

trary parameters. One use of this method by the RealMedia player (i.e. client) is to request a particular delivery bandwidth from the server. The details of how a decision is made to change the bandwidth and on how the server manages to adjust the bandwidth of an existing stream are not publicly known. However, by correlating the relevant RTSP SET_PARAMETER method instances with the packet arrival times at the probe point, we can observe the interaction from an *mmdump*-generated trace. One such example is shown in Figure 9 and explained below. (Note that RealMedia uses a proprietary transport protocol on top of UDP for media streaming. It was therefore not possible to monitor the sequence numbers of the media stream as would be possible for an RTP-based media stream.)

From the NYC trace data we extracted the control and data packets for a particular RTSP session for which we saw traffic in both directions between client and server. The session in question was streamed from a live source and contained only a single media stream. In Figure 9 we plot the timestamp of each UDP packet of the media stream as it was captured by *mmdump*. The total duration of the trace is 75 seconds. Time is on the X-axis while the Y-axis reflects the number of the corresponding packet. The slope of this plot is therefore an indication of the rate at which UDP packets where logged by *mmdump* (and the rate at which packets were sent by the source), with a steeper slope corresponding to a higher rate. The slope of the first part of the plot, packets 0 to 200, is clearly steeper than for the final part of the plot, packets 250 to 450.

Superimposed on the plot of UDP packet timestamps, is a number of horizontal dotted lines. Each horizontal line corresponds to the arrival of a SET_PARAMETER method for a bandwidth parameter as seen by the probe point. The value shown is the requested delivery bandwidth in bits per second. The sequence of these parameter requests goes from 21400 to 9361 to 17260 and 1977 in the first part, to 21400 and 13781 in the middle part and ends with a more modest sequence of 9163 to 7554 to 6621 to 12140 to 9870 and 6563 in the final part of the plot. This corresponds with the observed flatter slope of the last part of the plot. (Note that since the probe point is somewhere in the network between the client and the server, there will be a time lag between the time that *mmdump* records a SET_PARAMETER method, and the time that the server will have responded to it.)

4.2 H.323 Results

4.2.1 Individual session in controlled environment

First we first show how *mmdump* captures an H.323 session in a controlled environment. In the lab three machines are connected over a shared Ethernet link. Two Windows PC machines run Microsoft NetMeeting 3.1 and they make a video conferencing session with each other using the H.323 protocol. A third machine runs *mmdump* to capture the session on-line. The session lasts for approximately 35 seconds. Figure 10 shows packet arrival events grouped by channels. Each horizontal line indicates a channel, and there are five channels created in the duration of the session. Each small vertical line on a horizontal line indicates a packet arrival event. The session begins with the establishment of the Q.931 channel, followed by the H.245 channel. Then, NetMeeting uses H.245 to negotiate ports for three data channels, namely the TCP data channel, the RTP control channel and the RTP data channel. The latter two channels use UDP. In NetMeeting, the TCP data channel carries file transfer, chat, and whiteboard messages. The RTP data channel carries multimedia traffic such as voice and video. The RTP control channel carries metadata for the RTP data. Because this session exchanges video images in real-time, the bulk of the packets are RTP data, identified by the thick line of the RTP Data Channel.

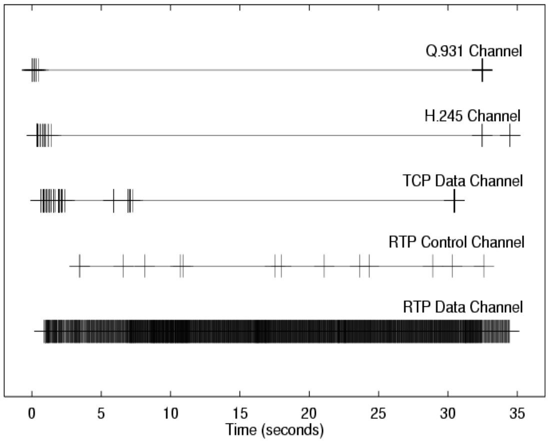

Figure 10: Packet arrival events for each channel of one H.323 session

It is important to note that except for the Q.931 channel, whose callee port number is well-known, the port numbers associated with the other 4 channels are dynamically negotiated. The callee port number of the H.245 channel is embedded in the CONNECT message of a Q.931 packet. The port numbers for the data channels are negotiated by the H.245 Open Logical Channel messages.

4.2.2 Sessions in the public Internet

Next, we present some H.323 results gathered over the public Internet. As in the RTSP case, the results presented here are from traces captured at the NYC probe point in AT&T's commercial IP network.

The trace analyzed in this section was started on Sunday August 22 1999 at 3:25pm EDT and lasted for 72 hours. We captured 2667 H.323 sessions containing 540MB of data. As in case of RTSP, we saved the entire length of TCP packets, but we saved only the first 136 bytes of UDP packets to reduce data size. Less than 1% of packets were lost in the kernel according to statistics produced by *tcpdump*.

There are two issues that may affect our result. First, the current implementation of the H.323 module assumes peer-to-peer communication. It does not work correctly if three or more parties are involved in a session. The module, however, is known to work with various types of H.323-enabled software, including Intel Video Phone, MediaRing GoldenEye, Microsoft NetMeeting, and VocalTec Telephony Gateway. Second, the traffic observed at the

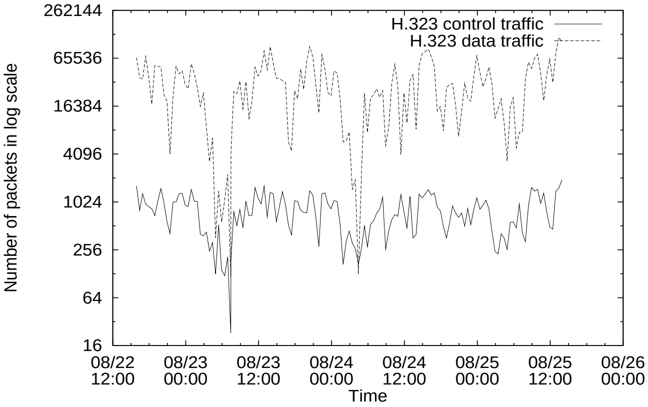

Figure 11: Packet counts for H.323 control versus data traffic

probe point is highly asymmetric. Therefore we incorporate various heuristics to the H.323 module so that it can track a session while seeing traffic flowing in only one direction. For example, if we only see the traffic from caller to callee and not from callee to caller, we will not get the CONNECT message sent by the callee, which contains the callee's port number to follow the subsequent H.245 channel. In this case, we guess that the H.245 port number of the caller is a small increment (one or two) of the caller's Q.931 port number. This appears to work reasonably well in practice.

Traffic Characteristics

Figure 11 shows the amount of aggregated H.323 control traffic (traffic exchanged in Q.931 and H.245 channels) and H.323 data traffic (traffic exchanged in H.323 data channels) over time. Not unexpectedly, the figure shows that the amount of control traffic is significantly lower than the amount of data traffic (note the logarithmic scale on the y-axis) and that there is a positive correlation between the amounts of control traffic and data traffic. Unexpectedly, there does not appear to be any notable pattern in the times of day when sessions occur. This is something we intend to investigate further.

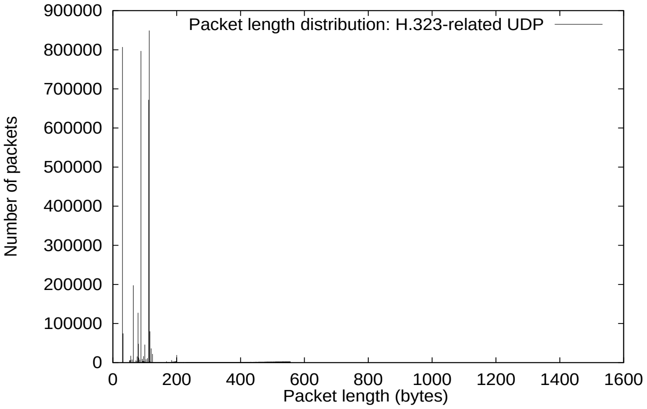

Figure 12: UDP packet length distribution for H.323 related traffic

Packet Length Distribution

Conferencing and packet telephony multimedia applications generally require good real-time performance. Therefore, we expect that these applications prefer to exchange smaller packets with higher packet rate rather than larger packets with lower packet rate. Here we show the packet length distribution for H.323 related UDP traffic in Figure 12. We observe that significant peaks are at packet lengths smaller than 200 bytes, which are shorter than typical MTU sizes.

As for the RTSP results, the packet length distribution for TCP traffic has a familiar distribution with a large peak at 40 bytes corresponding to TCP ACK, FIN, and SIN packets, and several peaks related to different MTU sizes.

Per-Session Statistics

One advantage of using *mmdump* is its ability to track each session individually. We will show an example that derives results based on per-session statistics.

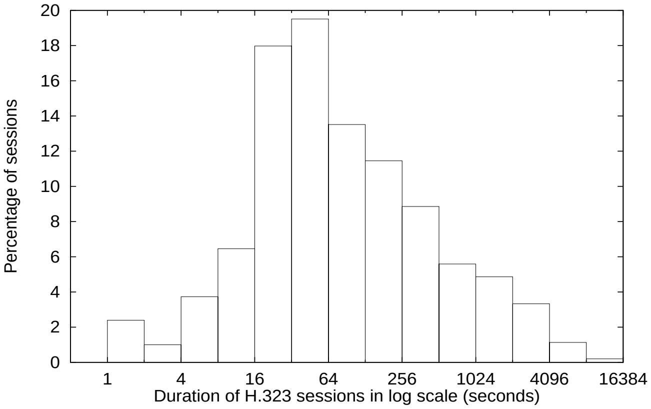

Figure 13: Duration of H.323 sessions

One question of interest is how long an H.323 session lasts. Figure 13 shows a histogram of different ranges of session duration with the percentage of sessions in that range. Here we consider only the subset of sessions for which *mmdump* was able to capture some UDP packets, and discard sessions for which *mmdump* did not capture any UDP packets. The latter can happen if the callee of a session did not answer or had incompatible terminal capabilities with the caller. Session duration is computed as the time between the first packet received (usually the Q.931 SETUP packet) and the last packet received (usually the H.245 FIN packet). The figure shows a majority of calls last between 16 seconds and four minutes. The figure also shows several sessions lasting longer than an hour.

5 Related Work

A recent paper [12] presents a preliminary analysis of streaming media traffic originating from a popular Internet audio service. It is one of the first studies of its kind. However, the set of IP addresses corresponding to the media servers under study was known a priori. In addition, the link under study was close to these servers and was known to carry all the traffic of interest. Under those conditions, it is not difficult to set up static packet filters to capture this traffic without overwhelming the trace collector with irrelevant traffic. That work therefore does not address the challenges of monitoring unknown multimedia traffic on an arbitrary link as ours does.

A large body of Internet traffic capture and analysis software has been developed over the years. Here we survey the subset that we feel is most relevant to our work.

The *tcpdump* [9] tool and its underlying packet capture library *libpcap* [8] have been widely used by the Internet research community. We have already described *tcpdump* in detail and noted that it does not handle dynamically negotiated port numbers. *mmdump* adds this capability to *tcpdump*.

Online extraction of application specific information, mainly to reduce the volume of generated data, has been reported in [5] and [10]. A software engineering approach similar to our own, is presented in [5] where *tcpdump* has been extended to perform online extraction of HTTP information. A more generic measurement platform, called Windmill, is described in [10]. This platform is meant to run continually providing the means to perform several "experiments" without ever terminating the Windmill instantiation. Since different experiments might be interested in different packet streams, the platform has the ability to dynamically modify the packet-filter expression. This change in packet filter expression is however performed at the time granularity of different experiments, not on the per-multimedia-stream timescales that *mmdump* deals with.

CoralReef [3] is an evolving suite of tools for collecting and analyzing Internet traffic. It is built upon the *libcoral* packet monitoring library and aims for flexibility and high performance. To our knowledge, CoralReef does not yet handle dynamically negotiated port numbers.

Narus [13] and Packeteer [14] have introduced commercial traffic capture and analysis products that reportedly handle dynamically negotiated port numbers. However, we have not had the opportunity to evaluate these products. To our knowledge, their internal details have not been made public and their source code is not available.

There are a number of tools tailored to monitoring and analyzing multimedia traffic. Among these are *rtpdump* [17] and *rtpmon* [1]. *rtpdump* decodes and displays RTP packets. *rtpmon* monitors RTP sessions and displays statistics based on the contents of RTCP packets. Neither tool parses session control protocols like RTSP and H.323, or handles dynamically negotiated port numbers.

We have focused on techniques for gathering information about multimedia sessions. We believe these techniques can be extended and applied to other related topics including monitoring session QoS, recording session duration and bandwidth usage for accounting purposes, and monitoring session activity for network intrusion detection purposes, e.g. by recording all FTP transfers.

6 Conclusions and Future Work

We have presented the design, implementation, and use of a new tool for monitoring multimedia traffic on the Internet. *mmdump* is based on *tcpdump* and further incorporates several novel features that make it practical to monitor multimedia traffic on an arbitrary link. One, it employs protocol-specific parsers to determine which port numbers are dynamically selected for media transport by multimedia session control protocols. Two, it maintains per-session state to record information such as session start/end times, media types, associated traffic. Three, it uses heuristics to deal with incomplete information due to asymmetric routing.

We have been using *mmdump* to monitor traffic from RTSP and H.323 sessions in AT&T's commercial IP network. The tool has already helped uncover a number of interesting features of this traffic:

- Multimedia sessions have a rich structure. We have seen examples of seemingly simple news clip presentations which are composed of more than 10 objects transferred over different port numbers and from multiple servers in different domains. As with web pages, this is partly due to the inclusion of advertising.
- Access patterns for multimedia objects follow distribution in which popularity drops off quickly outside a relatively small number of extremely popular objects. This has implications for caching.
- The RTSP protocol has a generic "SET_PARAMETER" method. In our measurements we observed that RTSP clients use this to request that servers adjust the transmission rate for ongoing sessions, based for example on observed packet losses. This finding begins to address the issue of whether multimedia traffic exhibits appropriate congestion-control behavior.
- The duration of H.323 sessions vary greatly, from a few seconds to over an hour. A majority of sessions last between 16 seconds and four minutes. This distribution of call durations is similar but not identical to that of traditional long-distance telephone traffic.

In terms of ongoing and future work, we have recently added to *mmdump* a different and more complete H.323 parser than the one described in this paper. We are experimenting with using it to monitor the quality of service in a commercial voice-over-IP trial. We have also developed a rudimentary SIP parser to add to the existing RTSP and H.323 parsers. In order to improve the performance of dynamic port processing, we are looking into adopting a modified BPF+ [2] that includes compiler support for incremental filter updates. Finally, we continue to use *mmdump* to monitor multimedia traffic on the public Internet and plan to perform a more thorough analysis of this traffic's growth and characteristics.

Acknowledgements

The RTSP parser was derived from a public-domain RTSP implementation by RealNetworks [16]. The initial H.323 parser was derived from software developed at Columbia University by Christo-

pher Kang. A later version of *mmdump* used the H.323 parser developed in the OpenH323 project (www.openh323.org).

References

[1] Bacher, D., Swan, A., and Rowe, L. A. rtpmon: A Third-Party RTCP Monitor. http://bmrc.berkeley.edu/people/drbacher/projects/mm96-demo/index.html.

[2] Begel, A., McCanne, S., and Graham, S. L. BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture. Proc. ACM SIGCOMM '99, August 1999.

[3] CAIDA. Coralreef. http://www.caida.org/Tools/CoralReef/.

[4] CNN. http://www.cnn.com.

[5] Feldmann, A. Continuous online extraction of HTTP traces from packet traces. Proc. W3C Web Characterization Group Workshop, November 1998.

[6] Handley, M., Schulzrinne, H., Schooler, E., and Rosenberg, J. SIP: Session Initiation Protocol. RFC 2543, March 1999.

[7] Recommendation H.323: Visual Telephone Systems and Equipment for Local Area Networks Which Provide a Non-guaranteed Quality of Service. ITU-T, 1996.

[8] Jacobson, V., Leres, C., and McCanne, S. pcap - Packet Capture library. UNIX man page.

[9] Jacobson, V., Leres, C., and McCanne, S. tcpdump - dump traffic on a network. UNIX man page.

[10] Malan, G. R., and Jahanian, F. An Extensible Probe Architecture for Network Protocol Performance Measurement. Proc. of ACM SIGCOMM'98, August 1998.

[11] McCanne, S. R., and Jacobson, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. Proc. 1993 Winter USENIX Technical Conference, January 1993.

[12] Mena, A., and Heidemann, J. An Empirical Study of Internet Audio Traffic. Proc. IEEE Infocom 2000, March 2000.

[13] Narus. http://www.narus.com.

[14] PACKETEER. http://www.packeteer.com/.

[15] RealNetworks. http://www.real.com.

[16] RealNetworks. RTSP: Reference Implementation. http://www.real.com/devzone/library/fireprot/rtsp/index.html.

[17] Schulzrinne, H. rtpdump. http://www.cs.columbia.edu/~hgs/rtp/rtpdump.html.

[18] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V. RTP: A Transport Protocol for Real-Time Applications. RFC 1889, January 1996.

[19] Schulzrinne, H., Rao, A., and Lanphier, R. Real Time Streaming Protocol (RTSP). RFC 2336, April 1998.

[20] W3C. SMIL: Synchronized Multimedia Integration Language. http://www.w3.org/AudioVideo/#SMIL.