The Architecture of CoralReef: An Internet Traffic Monitoring Software Suite

Ken Keys, David Moore, Ryan Koga, Edouard Lagache, Michael Tesch, and k claffy

I. Introduction

The volume and complexity of traffic on the Internet is increasing rapidly, making it both more difficult and more important to understand. To this end we have created the CoralReef passive traffic monitoring suite, which can be used by network users, administrators, and researchers to measure and analyze network traffic. In this paper, we will present the CoralReef design philosophy, overall architecture, and capabilities.

The CoralReef architecture is organized primarily into two "stacks" of software components, as shown in Figure 1, plus a few other small utility components. The *raw traffic stack* deals directly with individual PDUs (packets or cells) read from a network traffic stream, and the *flows stack* deals with traffic data that have been aggregated into "flow intervals". Each component of a stack builds upon the components beneath it. Any level of either stack may serve as a base upon which users may build their own software. The API provides many features to perform the most commonly needed operations, and many programming hooks to allow the programmer to customize the behavior of the library at any level. Except for the hardware-specific drivers, all of the software components are portable to most POSIX systems.

The core of the the raw traffic stack is the *libcoral* C library. In order to allow programmers to write a single application to access many types of data sources, *libcoral* provides a consistent API for capturing traffic from specialized ATM and POS capture cards from multiple vendors, as well as from pcap interfaces, while hiding the details of the hardware and drivers from users who do not need or want to see them. The same API is used to read packet capture files in any Coral format (including NLANR formats), pcap, and DAG [1] ATM and POS formats. All of these data sources appear the same to the user, and can even be read simultaneously. The *libcoral* API can operate on ATM cells, blocks of cells, or link layer packets, one at a time or via callbacks; the application developer can use whichever is most convenient. To facilitate rapid prototyping and development, another design goal is to provide the same API in C, C++, and Perl. Because the Perl module CRL.pm directly calls the C routines, Perl scripts using CRL.pm perform well enough for many practical applications.

Several additional components are included to perform auxiliary tasks as needed at any location in the stack structure. For example, there is a module for looking up IP address prefixes in Border Gateway Protocol (BGP) routing tables to find their autonomous system (AS) numbers, and another for determining geographic locations via NetGeo [2].

The flows stack includes modules for storage and manipulation of tables of frequently collected aggregate data. Measurements of traffic volume in bytes, packets, and flows can be aggregated by any combination of source and destination hosts, IP protocol, and ports. These modules provide methods to automatically aggregate data into tables with different keys and allow for efficiently selecting entries that represent the most traffic. For example, a single method call will convert a table of byte and packet counts aggregated by source AS number into a table aggregated by source country. Higher level applications are written using these building blocks; for example, a small Perl program is sufficient to create a traffic matrix by IP address or AS number, while larger programs (such as the realtime report generator t2_report) are built from complex arrangements of the same components.

II. Background and Related Work

To our knowledge, all existing tools for passive network workload characterization support features that cover only a subset of those covered by CoralReef. What makes CoralReef unique is that it supports a large number of features at many layers, and provides APIs and hooks at every layer, making it easier for anyone to develop new applications with a minimum of duplicated effort.

CoralReef began its existence as a library and drivers for monitoring ATM traffic from the same specialized hardware used by OC3MON [3]. For its research, CAIDA needed to monitor high volumes of ATM traffic simultaneously across multiple interfaces, each of which carried multiple streams of traffic (ATM virtual channels). OC3MON hardware and related components served these needs while few other tools did, but CAIDA research demanded more general software than that used in OC3MON.

Specialized hardware is essential for monitoring high volume traffic. Several commercial products, such as Narus IBI Platform [4] and Niksun NetVCR [5], use specialized monitoring hardware on a dedicated node to tap a network link. On the other hand, on links near the edge of the network where volume is lower, it is usually more economical and practical to use the network interfaces provided by

At the time of their work on CoralReef, all authors were with CAIDA, San Diego Supercomputer Center, University of California, San Diego. All authors except Michael Tesch are still with CAIDA at the time of this writing. E-mail: {kkeys,dmoore,rkoga,elagache,kc}@caida.org.

Support for CoralReef is provided by NSF grant NCR-9711092, DARPA NGI Contract N66001-98-2-8922, and by CAIDA members.

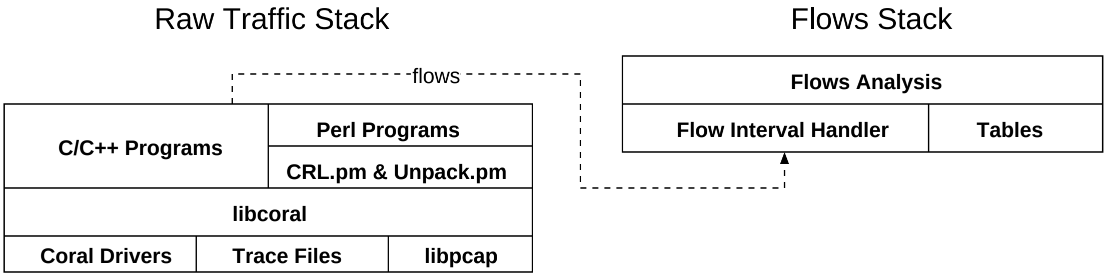

Fig. 1. Software components of CoralReef

the operating system on an existing node. Some tools can use an operating system's interfaces directly, and others use libpcap [6] to access interfaces. However, few network monitoring tools can make use of both specialized hardware and built-in network interfaces. CoralReef tools can use both, allowing their deployment anywhere on a network.

Many of the existing tools need to parse several layers of low-layer protocols just to reach the higher layer protocol in which they are actually interested. Operating systems have layer 2 parsers built into their network interface code, but not accessible to the programmer, and therefore useless for dealing with layer 2 data from other sources. CoralReef attempts to satisfy the need for a network analysis library that can parse layer 2 protocols.

Some tools, notably Ethereal [7], Narus IBI Platform, and libnids [8], parse layer 4 and higher protocols. All of these perform TCP stream reassembly. CoralReef does not currently provide an API for layer 4 reassembly.

CoralReef includes programs for generating flow data, APIs for manipulating them, and tools for flow-based report generation. Both NeTraMet [9] and NetFlow export [10] generate flow data. A powerful feature of NeTraMet is that it supports a programming language for defining flows. Because NetFlow export is supported by multiple router vendors, it provides a convenient way to obtain flow data without adding additional passive monitoring equipment to the network. Cflowd [11], combined with the arts++ [12] library, collects and aggregates NetFlow data from routers and provide a programming API for tables of flows and text reporting functionality. Tools for graphical web-based flows reporting, in addition to those in CoralReef, include FlowScan [13] and Fluxoscope [14].

III. Components of the Raw Traffic Software Stack

A. Input Sources

CoralReef can read traffic data from custom device drivers, trace files, and libpcap.

A.1 Device Drivers

CoralReef device drivers allow passive collection of data from specialized collection hardware. CoralReef includes FreeBSD drivers for Apptel POINT[1] (OC3 and OC12 ATM) and FORE ATM (OC3 ATM) cards, and supports[2] Linux drivers for WAND DAG (OC3 and OC12, ATM and POS) cards.

All of the supported cards write traffic data into blocks of memory, and generate an interrupt after filling a block. CoralReef drivers use a block size of approximately 1 megabyte. Because of the large block size, the system does not spend an undue amount of time handling interrupts. Large blocks can also be written to disk more efficiently than smaller ones.

A.2 Files

The ability to read traffic data from files allows CoralReef applications to analyze traffic offline, and on hosts other than that from which the data were collected. CoralReef files contain version information that allows the software to read data from files written by any past version of itself. It can also read files written by NLANR [15] or MCI [3] OC3MON software and dagtools [1] software. CoralReef can identify NLANR and MCI files automatically by the format of their header, but must be explicitly told that a given file is in dagtools format.

A.3 Libpcap

While CoralReef was originally designed to analyze traffic from ATM devices, it was apparent that its IP analysis features would be useful on IP traffic carried over any physical medium or datalink protocol. By supporting libpcap as a back end, the CoralReef library allows all IP applications built on it to work on a large number of widely available network interfaces with no change to the applications.

B. Libcoral

Libcoral is a C library that does the bulk of the work involved in reading passive traffic from any of the sources described in Section III-A. It is the base on which all CoralReef applications are built. Applications that use *libcoral* see a uniform programming interface to all supported source types, so they do not need to be rewritten for each input source. In the future, when *libcoral* supports new network card and monitor systems, existing application programs that use *libcoral* will be able to use the new sources with little or no change.

[1] Apptel has discontinued production of the POINT cards.

[2] The DAG drivers are not included in CoralReef, but are part of the DAG software package [1].

In CoralReef terminology a *source* is a device, network interface, or file from which data are read. The concepts of *source* and *interface* are kept separate for several reasons. First, a trace file may contain data collected from multiple interfaces. This is often the case when data were captured from a pair of ATM devices that monitored both directions of traffic. Second, the format of data from a particular type of device is the same whether read directly from the device or from a file created by recording data from the device. Separation of format handling from I/O handling allows the same format handling code to be used for a device and for a file made from that device, and allows reading files on platforms that do not support the devices from which the files were made.

B.1 Sources

Internally, *libcoral* organizes source types into *source type switches* that contain information on how to handle each source, including:

- *is_live* — whether the source is a file or live interface
- *is_buffered* — whether the source does its own buffering
- *is_coral_dev* — whether the source is a CoralReef device
- *init()* — initialize the source
- *read_min()* — read the minimum amount of data from a specific interface of the source
- *nextblk()* — read a block of data from the source
- *release()* — free a queued block
- *has_buf()* — see if source has readable buffered data
- *stop()* — stop capturing on the source
- *close()* — close the source

For the CoralReef device sources, the *init* function performs the `ioctl` requests to upload firmware to the device and initialize it according to requested parameters, and creates a single interface structure initialized to the appropriate *iface type switch* and other settings. The *nextblk* function performs the `ioctl` request to get a Coral data block pointer from the driver. All CoralReef device sources use the same *read_min* function, `coral_blk_read_min`, which calls *nextblk* and stores the Coral data block pointer and other information on the interface structure. Since CoralReef device sources do not queue blocks nor allow reads of partial blocks, they do not have *has_buf* or *release* functions.

CoralReef files begin with a header that describes each interface which produced the data in the file, including the interface's type and configuration options. These files may contain data collected from any type of Coral device: POINT, FORE, or DAG. The *init* function for CoralReef files reads this information and uses it to create and initialize an interface structure for each interface recorded in the file. The *nextblk* function reads a Coral data block from the file into an allocated buffer. The *read_min* function calls *nextblk* on the source until it gets a block for the desired interface. But since a file may contain data from multiple interfaces, *nextblk* may return a block from an interface other than the one requested by *read_min.* Each time this happens, *read_min* queues the block on the interface structure to which the block belongs, avoiding the need to seek within the file when data from the other interface are needed later.

Unlike CoralReef files, files created by the dagtools software do not contain CoralReef file headers or other identifying information, so *libcoral* must be told explicitly that a given file is a dagtools file and what parameters were used to record it. The *init* function creates an interface structure and initializes it to DAG values and the user-specified parameters. Coral block headers and boundaries are also absent from files written by dagtools software, so *nextblk* simply reads 1 MB worth of records into a buffer and treats it as a Coral block. Thus the dagtools file switch can use the same function as the Coral file switch for the *read_min* entry point.

The *init* function for pcap file and pcap live sources calls the corresponding pcap initialization function and creates a single interface structure initialized to the appropriate *iface type switch* and other settings. Pcap sources do not have Coral blocks, and thus no *nextblk* function. Both types of pcap sources use the same *read_min* function, `coral_pcap_read_pkt`, which reads a single packet via `pcap_dispatch` and stores a pointer to it on the interface structure.

B.2 Interfaces

Libcoral organizes information about handling interface types into an *iface switch* for each type. Interface type information includes a description of the record layout and functions to normalize timestamps and construct packets from the data stored on the interface structure by the source functions.

The record layout is used for interfaces that have Coral data blocks. It describes the size of the records and the locations of fields within the records. Access to fields of different record formats is abstracted by macros that find the fields using the record layout information in the *iface switch*, hiding the layout details from the application programmer.

Each interface type has its own timestamp format. Functions on the *iface switch* convert a timestamp to a canonical form, `struct timespec`, so higher level code does not have to deal with the different formats. These functions can also (optionally) automatically correct timestamp errors generated by the device, e.g., the failure of the FORE firmware clock to increment immediately upon a hardware clock wrap. This error, if left uncorrected, would manifest itself to higher software layers as a backward time jump of approximately 2.6 ms followed a few cells later by a forward jump of similar magnitude.

The *prep_pkt* function prepares a (possibly incomplete) packet from the data returned by the source functions. For POS and pcap interfaces, preparation is trivial, since these interfaces return packets. But for ATM interfaces, preparing a packet means performing AAL5 reassembly for each virtual channel from the ATM cells in the Coral data blocks returned by the source. Normally, reassembly requires appending a copy of each ATM cell to a buffer associated with the virtual channel until the end of the AAL5 PDU (protocol data unit) is found or the maximum number of captured cells per PDU are seen. But in the common case where only one cell per PDU is captured, the reassembly function can skip the expensive copy; the resulting packet structure will point directly to the original cell contents instead of to a buffer full of concatenated cell copies.

B.3 Traffic Reading Functions

There are three ways to read traffic from a CoralReef source: by block, by cell, or by packet. Each of these methods can read data from multiple sources simultaneously.

Block reading works only on sources that operate on Coral data blocks. It is implemented by calling *nextblk* on a source whenever `select()` indicates that the source is readable. Block reading is the most primitive interface and is only used by a few applications, most notably *`crl_trace`*, which simply writes the blocks to a CoralReef trace file.

Reading by cells works only on ATM sources, and is useful for applications that do ATM layer analysis; e.g., measuring traffic volume per virtual channel. Cell reading functions manage Coral data blocks internally, and dole cells out to the caller as requested.

The most widely used type of reading is by packet. *Libcoral*'s packet API hides the details of the physical layer from the application programmer. In this context, "packet" means a layer 2 PDU, e.g., an Ethernet frame or AAL5 PDU. The resulting packet structure includes a subinterface identifier so the caller can distinguish separate data streams within an interface. Currently, the only supported type of interface that has subinterfaces is ATM, for which the subinterface identifier is the VPI and VCI. The result may also include pointers to the header and trailer of lower layer protocols (e.g., ATM cell header and AAL5 trailer) so the caller can access them if desired.

Traffic analysis applications usually need the raw data in temporal order, and often need to perform periodic operations at fixed intervals, and stop processing after a specific duration. To make application programming easier, *libcoral* can take care of all these needs. Each interface returns its own data in temporal order, but low-level buffering on the interfaces means data read from multiple network interfaces will not be in the correct order. Reading from multiple interfaces is common on ATM network links, which have separate interfaces for each direction of traffic. So, the *libcoral* cell and packet reading functions have the option to automatically merge-sort the records they return to the caller. Merge-sorting requires that *libcoral* have at least some data from each interface; *read_min* guarantees that this requirement is met.

If the programmer specifies an interval, the *libcoral* traffic reading functions compare the data timestamps to the interval, and return control to the caller when they encounter a timestamp beyond the end of the interval. Similarly, the programmer can specify a duration, and the reading functions will not return data with timestamps that exceed the duration. All of these time features use data timestamps generated by the source instead of the CPU clock so they are not affected by buffering between the source and the software, and even work on file sources. *Libcoral* also uses the CPU clock to test duration, so even if all interfaces are silent or buffering too long and thus not providing timestamps for comparison, *libcoral* will detect the duration expiration and correctly stop reading.

Protocol parsing is another common feature required by traffic analysis applications, so we put this feature into *libcoral*. The "packets" returned by the packet reading functions are usually layer 2 PDUs, which may contain multiple sublayers of encapsulation. The `coral_get_payload` function parses the encapsulating protocol and returns a pointer to the payload. If the encapsulating protocol indicates that the payload contains another sublayer recognized by *libcoral*, the function also returns the identity of its protocol. For convenience, *libcoral* also includes `coral_get_payload_by_layer` and `coral_get_payload_by_proto`, which recursively call `coral_get_payload` on each enclosed PDU until it finds one at a specified OSI layer or with a specified protocol, respectively. The `coral_get_payload_by_proto` function is most commonly used to easily find the IP layer, skipping all lower sublayers. For example, a PPP packet read from an interface might contain a BRIDGED_LAN packet, which might in turn contain an IEEE 802.3 packet, which finally contains an IP packet; a programmer could reach the IP packet with a series of conditions and calls to `coral_get_payload` or with a single call to `coral_get_payload_by_proto`. These functions can optionally print the protocol information they parse, much like *tcpdump* [16] (although they do not understand as many protocols as *tcpdump*). One major advantage of *libcoral* over *tcpdump* is that *libcoral* provides an API to access the protocols at any layer, instead of just printing what it finds. Protocols recognized by *libcoral* include ATM_RFC1483 [17]; LANE for IEEE 802.3/Ethernet [18]; Ethernet [19]; IEEE 802.3 [20]; Cisco HDLC; PPP [21] over HDLC [22], [23] or Ethernet [24]; bridged LAN (over PPP) [25]; IP [26]; ARP [27]; ICMP [28]; TCP [29]; and UDP [30]. If *libcoral* can not determine the lowest level protocol of traffic on a device from the device itself, the user must tell *libcoral* via a command line option or configuration file. The user may configure each virtual channel of an ATM interface as having a different protocol.

Another feature of *libcoral* frequently needed by traffic analysis applications is traffic filtering. When reading from ATM interfaces, *libcoral* can filter by VPI and VCI, as configured by the user. Filtering is useful for eliminating signaling channels or other uninteresting traffic at an early stage. Using libpcap, *libcoral* can also filter packets with BPF [31] filter expressions. Most filter expressions need only the IP and transport header, which usually fit within the first ATM cell of an AAL5 PDU unless IP options are used. McCreary [32] has observed that only 0.003% of IP packets contain options. Since the first cell usually contains sufficient information for the filter, *libcoral* can apply the BPF filter to the piece of the packet contained in the first cell before doing AAL5 reassembly. To do this, *libcoral* uses a slightly modified BPF interpreter that indicates not just "pass" or "fail," but also "packet was too short to tell." In the last case, typically caused by IP options pushing the transport header out of the first cell, *libcoral* will reapply the filter against the fully reassembled packet. When the filter drops a large fraction of the packets before reassembly, *libcoral* avoids a significant amount of unnecessary reassembly overhead.

There are many configurable parameters for using *libcoral*: protocol specifications for interfaces, interval, duration, length of packet to capture, filters, diagnostic verbosity, etc. *Libcoral* has functions to read these options from the command line or a configuration file. These functions save work for the application programmer, and presents application users with a means of configuration that is consistent across all applications.

C. Perl API

The Perl module CRL.pm provides a Perl interface to *libcoral* via the SWIG [33] interface generator, allowing programmers to call its functions from Perl. It also includes several convenience functions for performing common actions, such as opening all input sources and setting default reading options, or extracting IP-level data from a packet or cell. CRL.pm uses the *libcoral* C library as a back end because much of its functionality would be difficult or impossible to implement in Perl, and implementing the rest of it would be a needless duplication of effort.

While programmers using *libcoral* can simply use C structures to access packet headers and other CoralReef-specific structures, native Perl techniques to access these structures are not sufficient to deal with the large volumes of data encountered in traffic analysis. The Unpack.pm Perl module provides a Perl interface to these structures through SWIG, which is both more convenient and several times more efficient than a Perl implementation would be. The Unpack library gives the programmer the ability to access individual fields in packet data, and can also do other useful field manipulations, such as converting IP addresses from binary to a string format.

D. Raw Traffic Applications

All raw traffic applications included in CoralReef have names beginning with "`crl_`", and accept the common configuration options defined by *libcoral*. CoralReef includes applications to capture raw PDUs to a file, convert trace files to other formats, filter traffic, print packets, analyze timestamps, monitor for security problems, collect DNS usage statistics, and more. One of the applications, *`crl_flow`*, aggregates raw traffic data into flow data for use by the flows software stack. We will describe it further in section IV.

IV. Components of the Flows Software Stack

While the raw traffic stack provides access to network data in the most detailed manner, often one does not need so much detail. To make certain kinds of data more manageable, CoralReef has a flows stack that aggregates data across time based on the *flow* to which they belong.

The CoralReef flows stack has many container objects, discussed below. Figure 2 shows the relationships among the objects and concepts used in the flows stack.

A *flow* is loosely defined as the set of packets that have common values of certain network related fields (keys). The specific keys of interest vary with the type of analysis, but are usually a subset of IP addresses, port numbers, IP protocol, AS numbers, network prefixes, and geographic locations (all except IP protocol apply to the traffic's source, destination, or both). A specific set of keys defines a specific *FlowType.*

Information about an individual flow is stored in a set of values called a *Counter.* In the most common case these values are just cumulative counters of the number of packets and bytes seen in a flow. However, a *Counter* may also contain averages and other more generalized types of values. A specific set of values used in a Counter and an "add" function for updating those values defines a specific *CounterType.*

The flows stack stores a set of flows and their associated Counters in a *Table.* This generic Table is the main data structure used by the flows stack. A specific type of a Table, defined by the FlowType and CounterType it contains, is called a *TableType.*

To make flow data management easier and to allow meaningful comparison of aggregated data, the flows stack organizes groups of Tables into *Intervals.* An Interval is a container for all flows on all interfaces and subinterfaces for a fixed time period. Within an Interval there is a Table for every combination of (sub)interface and TableType being collected or processed. In this manner, an Interval represents a snapshot of all collected network traffic for that time period.

The CoralReef flows stack is composed of programs that generate Intervals of Tables of flows, and libraries for reading, writing, and processing them.

A. Flow Generation

Flows can be created using CoralReef in many different ways, but the most commonly used flow generation program is *`crl_flow`.* This program produces flows based on a FlowType of a 5-tuple of source IP address, destination IP address, IP protocol, source port, and destination port. For ICMP packets, the type and code are used instead of the source and destination ports. Additionally, *`crl_flow`* records when the source and destination ports are not known, either because they are not meaningful for the given protocol or because they were not captured. A situation where port information is not captured occurs, for example, when only the first ATM cell is captured and IP options push the inner layer 4 PDU header beyond the first 48 bytes. The information recording that the ports are undefined is kept in an additional field of the flow, called `ports_ok`, which sometimes leads to flows being referred to as 6-tuples since they have 6 fields in the FlowType. The *`crl_flow`* program can also generate flows based on a FlowType of the IP packet length, which it outputs as a separate Table. By using the packet length FlowType, a

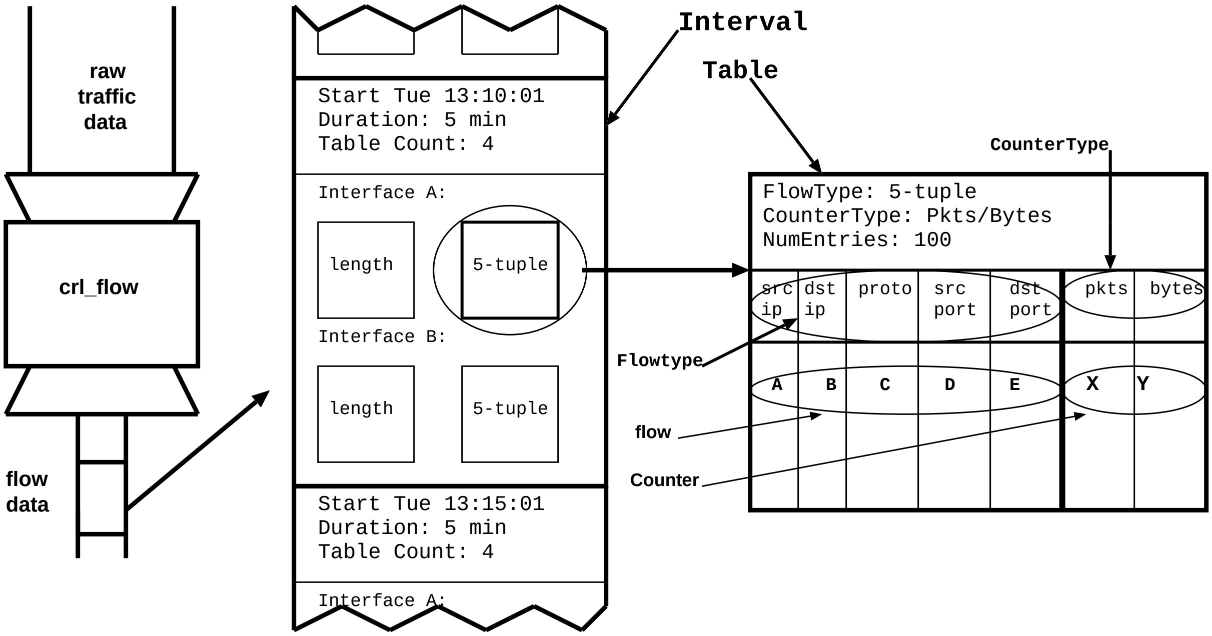

Fig. 2. Objects and concepts used in the Flows Stack

user can readily determine the number of packets and bytes seen for different sized packets. The CounterType that *crl_flow* uses to record flow data contains normal counters for the number of packets and bytes in a flow, and two fields that are not really counters at all: the timestamps of the first packet and most recent packet in a flow.

There are four primary methods of expiring flows: fixed timeout, adaptive timeout, protocol based, and time bucketed. These methods may also be combined to differing extents. With a *fixed timeout* method [34], a flow is expired when some fixed time period has elapsed since a packet was seen matching this flow (e.g., waiting 60 seconds after seeing the most recent packet). *Adaptive timeout* [35] is like fixed timeout, but the time period is dynamically chosen for this flow based on its previous behavior (e.g., waiting 6 times the average interarrival time of packets previously seen in this flow [36]). In *protocol based* expiry, a flow is expired due to observation of a protocol specific message (e.g., TCP FIN or RST). *Time bucketed* expiry expires all flows on a fixed time interval that is external and independent of the traffic (e.g., every five minutes from the start of the trace). Expiring with a time bucketed method is useful when the amount of traffic on a link in a given time window must be known accurately. Without time bucketed expiry, the traffic rate of a flow must be averaged over its entire lifetime to make time series plots of link utilization [13], [14]. When average rates are used to compute total traffic values, it is possible for the totals to exceed the link capacity or produce other unexpected results. In general, time bucketed expiry is more useful when studying the traffic characteristics of a *link* over a specific period, and non-time bucketed expiry is more useful when studying the characteristics of individual *flows*.

The *crl_flow* program currently supports fixed timeout, some specific kinds of adaptive timeout, and time bucketed expiry. For the fixed timeout expiry, the user specifies the number of seconds to wait. Time bucketed expiry uses the interval functionality of the raw traffic stack to periodically expire all flows. There are two adaptive timeout mechanisms, each of which takes two parameters. The first is based on a new adaptive timeout mechanism from NeTraMet [36], in which a flow is not expired at all during the first N seconds, but after that point a packet must arrive within M times the average interarrival time previously seen in that flow. The second adaptive timeout mechanism requires that a packet arrive within M times the last seen interarrival time, which defaults to N seconds when only one packet has been seen.

Once generated, flows are usually further analyzed with tools written using the Interval processing APIs.

B. Interval Handler

The CoralReef flows stack groups flow data into fixed time periods called *Intervals.* The time period for an Interval is used for reporting of flows, and need not be related to timeouts used for flow expiry. An Interval contains data for all flows that expired in the time period, and optionally for flows that are still active at the end of the time period. The sets of flows are each held in a *Table*, with separate Tables for active and expired flows, per (sub)interface and per TableType. For example, when reporting both active and expired flows in two TableTypes (e.g., 5-tuples and IP packet lengths) on an ATM link with 10 virtual channels there would be up to 40 Tables per Interval.

Within an Interval, there is meta-information describing the interval's beginning time and duration, as well as additional meta-information specific to each (sub)interface it contains. Meta-information for each (sub)interface includes the number of IP packets, IP bytes, and PDUs with unknown encapsulation, as well as timestamps for the first and most recent IP packet seen.

To improve efficiency in reading and processing, the meta-information for an Interval appears at the beginning, and is followed by the Tables of flows in the same order as specified in the meta-information. If any of the (sub)interfaces contain more than one TableType, then all of the Tables for that (sub)interface are grouped together.

The CoralReef flows stack provides APIs for reading, writing and manipulating Intervals. Operations provided by the Interval APIs include:

- reading or writing an entire Interval of Tables
- reading or writing an Interval incrementally
- reading only a requested subset of TableTypes from a file
- automatically creating a requested TableType that is not in the file if there is a Table from which it can be derived
- accessing Interval meta-information
- accessing (sub)interface data
- accessing Tables within Interval (sub)interfaces

By moving commonly performed data manipulation operations into a library, the CoralReef flows stack simplifies the design of flows analysis tools. In providing a mechanism, Intervals, for organizing and processing flow data, CoralReef makes processing, analyzing and storing flow data convenient and efficient.

C. Tables and Counters

In the CoralReef flows stack, a *Table* is a container for flows and associated data values, called Counters. In addition to simple gettor/settor operations, the Table API provides functionality for data aggregation and report generation.

Tables of a given TableType can be converted easily to another TableType by transformation of the flow keys. The associated Counters are combined to produce the aggregated result. For most CounterTypes this just involves adding the respective fields of the combining Counters. The simplest Table aggregation is the removal of key fields from the FlowType. For example, a 5-tuple Table can be aggregated directly to an IP-matrix Table, which uses source and destination IP addresses as its keys. More complicated aggregations available through CoralReef include mapping IP addresses to AS numbers via a routing table, and mapping IP addresses or AS numbers to countries.

On most production networks, the number of flows seen will be large for any time period longer than a couple seconds, thus making it difficult to pick out more important or interesting flows. To help identify "large" flows, the Table API has sorting functions which select flows based on Counter fields. For example, using a simple CounterType with sums of packets and bytes, selecting the 25 largest flows by bytes is a single operation.

General operations supported in the Table API include:

- inserting a new flow with an initial Counter
- updating the Counter of an existing flow
- extracting the Counter associated with a flow
- walking all of the flows in the Table
- sorting the flows by fields in their Counter, in increasing or decreasing order
- sorting partially by Counter fields ("top N" or "bottom N" flows)
- adding two Tables of the same TableType
- aggregating into a new TableType by a transform of the flow keys
- accessing a "total" Counter for the Table which is a sum of the Counters for all contained flows
- writing or reading a Table directly as text or binary

At the time of this writing, the Table API is only fully available within Perl. However, there are C++ backends for 18 common FlowTypes and a basic Counter that counts packets, bytes, flows and timestamps[3]. Other FlowTypes or CounterTypes are handled entirely in Perl. Work is ongoing to provide STL versions of Tables for use by C++ tools.

V. Other Software Components

CoralReef includes a stand-alone library called ASFinder, which maps IP addresses to AS numbers and network prefixes. The API is in Perl, but the actual search code is written in C++ and utilizes a Patricia trie, which makes it both fast and memory-efficient. In order to make these mappings, ASFinder must first have loaded a file containing a sorted list of network prefixes and AS numbers.

To create input files for ASFinder, CoralReef includes scripts that convert the output of a Cisco router's "`sho ip bgp`" command or the output of mrtd [37] into the format ASFinder uses.

Also included in CoralReef is the NetGeoClient [2] library, which maps AS numbers and IP addresses to their geographic locations. These are used by several CoralReef applications, notably ***t2_ASMatrix***.

VI. Conclusion

While CoralReef comes with many traffic analysis applications, its biggest strength is the flexibility and power of its libraries and components, which make it an excellent base upon which to build new applications. Some examples of actual CoralReef use are described below.

Published research that has taken advantage of CoralReef includes studies of characteristics of fragmented IP traffic [38] and trends in wide area IP traffic patterns [32]. Current work at CAIDA using CoralReef includes studies of distributed denial of service attacks, use of DNS root servers, and prevalence of plaintext passwords.

At least two existing traffic analysis applications, NeTraMet [9] and Vern Paxson's Bro [39], have been adapted to use the *libcoral* API to run on specialized ATM and POS hardware that they did not previously support.

CAIDA uses CoralReef continuously to generate real-time graphs of traffic loads [40].

CoralReef has also been used as a teaching tool in networking classes by Evi Nemeth and Geoff Voelker and on the CAIDA Traffic Analysis Teaching CD [41].

[3] C++ backends for Tables and Counters are currently only available to CAIDA members

The CoralReef package is available for download at `http://www.caida.org/tools/measurement/coralreef/`.

VII. Acknowledgments

We would like to thank Sean McCreary of the University of Colorado at Boulder (formerly CAIDA) for his work on BGP table parsing and other help; Jambi Ganbar of MCI (formerly CAIDA) for early prototypes and testing; Tony McGregor of the University of Waikato for his work on traffic analysis teaching materials using CoralReef; Sue Moon of Sprint Advanced Technology Laboratories for her feedback on CoralReef; Colleen Shannon (CAIDA), Young Hyun (CAIDA), Judy Trummer (UCSD) and David Meyer (Cisco Systems, Inc./University of Oregon) for their feedback and assistance with this paper; and everyone else at CAIDA for their input and support.

References

[1] Waikato Applied Network Dynamics group, "The DAG project," `http://dag.cs.waikato.ac.nz/`.

[2] David Moore, Ram Periakaruppan, Jim Donohoe, and kc claffy, "Where in the world is netgeo.caida.org?," in *INET 2000 Proceedings*, June 2000.

[3] J. Apisdorf, k claffy, K. Thompson, and R. Wilder, "OC3MON: Flexible, affordable, high-performance statistics collection," in *INET'97 Proceedings*, June 1997, `http://www.isoc.org/isoc/whatis/conferences/inet/97/proceedings/F1/F1_2.HTM`.

[4] Narus, "Narus IBI Platform," `http://www.narus.com/ibi/`.

[5] Niksun, "NetVCR," `http://www.niksun.com/products/netvcr.html`.

[6] S. McCanne, C. Leres, and V. Jacobson, *libpcap*, Lawrence Berkeley Laboratory, Berkeley, CA, available via anonymous ftp to ftp.ee.lbl.gov.

[7] Gerald Combs et al., "Ethereal — a network protocol analyzer," `http://www.ethereal.com/`.

[8] Rafal Wojtczuk, "libnids: network intrusion detection system e-box library," `http://www.packetfactory.net/Projects/Libnids/`.

[9] N. Brownlee, "RFC 2123: Traffic flow measurement: Experiences with NeTraMet," Mar. 1997, Status: INFORMATIONAL.

[10] "Cisco NetFlow," `http://www.cisco.com/warp/public/732/netflow/`.

[11] Daniel W. McRobb, "cflowd: Traffic flow analysis tool," `http://www.caida.org/tools/measurement/cflowd/`.

[12] Daniel W. McRobb, "arts++: Network data storage library," `http://www.caida.org/tools/utilities/arts/`.

[13] Dave Plonka, "FlowScan: A network traffic flow reporting and visualization tool," in *LISA Winter 2000 Conference Proceedings*. University of Wisconsin, Madison, Dec. 2000, USENIX LISA.

[14] Simon Leinen, "Fluxoscope: a system for flow-based accounting," `http://www.tik.ee.ethz.ch/~cati/deliv/CATI-SWI-IM-P-000-0.4.pdf`.

[15] "NLANR MOAT PMA — passive measurments and analysis," `http://moat.nlanr.net/PMA/`.

[16] V. Jacobson, C. Leres, and S. McCanne, *tcpdump*, Lawrence Berkeley Laboratory, Berkeley, CA, June 1989, available via anonymous ftp to ftp.ee.lbl.gov.

[17] Juha Heinanen, "RFC 1483: Multiprotocol encapsulation over ATM adaptation layer 5," July 1993.

[18] ATM FORUM Technical Committee, *LAN Emulation Over ATM Version 2 — LUNI Specification*, July 1997.

[19] Digital Equipment Corporation, Intel Corporation, and Xerox Corporation, *The Ethernet — A Local Area Network: Data Link Layer and Physical Layer (Version 2.0)*, Nov. 1982.

[20] IEEE, *IEEE Std. 802.3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, 1985.

[21] W. Simpson, "RFC 1661: The point-to-point protocol (PPP)," July 1994.

[22] W. Simpson, "RFC 1662: PPP in HDLC-like framing," July 1994.

[23] A. Malis and W. Simpson, "RFC 2615: PPP over SONET/SDH," June 1999.

[24] L. Mamakos, K. Lidl, J. Evarts, D. Carrel, D. Simone, and R. Wheeler, "RFC 2516: Method for transmitting PPP over Ethernet (PPPoE)," Feb. 1999.

[25] F. Baker and R. Bowen, "RFC 1638: PPP bridging control protocol (BCP)," June 1994.

[26] J. Postel, "RFC 791: Internet Protocol," Sept. 1981.

[27] D. C. Plummer, "RFC 826: Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware," Nov. 1982.

[28] J. Postel, "RFC 792: Internet Control Message Protocol," Sept. 1981.

[29] J. Postel, "RFC 793: Transmission control protocol," Sept. 1981.

[30] J. Postel, "RFC 768: User datagram protocol," Aug. 1980.

[31] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *USENIX Winter 1993 Conference Proceedings*. Lawrence Berkeley Laboratory, Dec. 1992, pp. 259–269.

[32] Sean McCreary and k claffy, "Trends in wide area IP traffic patterns: A view from Ames Internet Exchange," in *ITC Specialist Seminar on IP Traffic Modeling, Measurement and Management*, Sept. 2000.

[33] David M. Beazley, David Fletcher, and Dominique Dumont, "Perl extension building with SWIG," in *O'Reilly Perl Conference 2.0*, Aug. 1998.

[34] K. Claffy, H.W. Braun, and G. C. Polyzos, "Internet traffic flow profiling," Nov. 1993, `http://www.caida.org/outreach/papers/itf.html`.

[35] Bo Ryu, David Cheney, and Hans-Werner Braun, "Internet flow characterization - adaptive timeout and statistical modeling," in *PAM2001 — A workshop on Passive and Active Measurements*. HRL Laboratorie and NLANR, Apr. 2001, RIPE NCC.

[36] Nevil Brownlee and Margaret Murray, "Streams, Flows and Torrents," in *PAM2001 — A workshop on Passive and Active Measurements*. CAIDA, Apr. 2001, RIPE NCC.

[37] "MRT — multi-threaded routing toolkit," `http://www.mrtd.net/`.

[38] Colleen Shannon, David Moore, and k claffy, "Characteristics of fragmented IP traffic," in *PAM2001 — A workshop on Passive and Active Measurements*. CAIDA, Apr. 2001, RIPE NCC.

[39] Vern Paxson, "Bro: A system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23–24, pp. 2435–2463, Dec. 1999.

[40] "CoralReef demos," `https://anala.caida.org/CoralReef/Demos/`.

[41] Tony McGregor, "CAIDA traffic analysis teaching CD," http://traffic.caida.org/.