Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computer Networks 50 (2006) 2564–2576
Computer Networks
www.elsevier.com/locate/comnet

ANQL – An Active Networks Query Language ☆

Craig Milo Rogers *

Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292, United States

Available online 2 May 2006

Abstract

This paper explores the application of a relational database model to network communication processing. Packets are constructed of protocol data units, usually organized as an encapsulating sequence of protocol headers. These protocol headers may be modeled by relational database tables. Critical information about the state of a protocol processing node, such as connectivity and adjacency, link delay, and cpu utilization, may also be modeled through relational tables. An application-specific language, ANQL (Active Networks Query Language), is introduced to explore the database metaphor for packet processing. ANQL has been demonstrated in Active Network control and management plane activities, but ANQL or related languages might also be utilized as a full-fledged active packet language. ANQL is applicable in both event-driven and background processing environments, and may be used in centralized data collection and analysis processes, or in distributed implementations of packet analysis activities.

Keywords: ABone; Active networks; ANQL; ASP; NodeOS; SQL

1. Introduction

There are many actions that may be taken when processing packets in the nodes of a data communication network. Some actions are based on decisions that must be made on each packet as it trasverses each network node, according to policies specified for this packet by the originator of the packet (possibly as modified by previous nodes in the packet's traversal of the network), while other actions are made to implement policies that support the control and management of the communication network as a whole.

Examples of actions that may be taken include:

- routing packets towards their final destination based on criteria such as minimizing packet latency, maximizing overall throughput, or picking the most reliable or secure end-to-end path,
- checking for faults, and diagnosing their possible causes,
- detecting intrusions, and sending an alarm when a sequence of packets indicates that an attack is in progress,
- monitoring for undesirable payload content, such as computer viruses,
- determining when excessive traffic of a particular class is monopolizing network resources,
- reporting bandwidth utilization statistics.

☆ This work was supported in part by the Defense Advanced Research Projects Agency under contract BT6399C0032.
* Tel.: +1 310 448 9127; fax: +1 310 823 6714.
E-mail address: rogers@isi.edu

1389-1286/$ - see front matter
doi:10.1016/j.comnet.2006.04.002

The need to perform actions based on decisions and policies, such as those above, has been present since computer networks first became available. Many approaches to implementing them have been demonstrated in the past. The advent of Active Networking [15] [13] gives us a new set of tools for the implementation, control and management of computer networks, providing opportunities to create better solutions to long-existing network design problems.

Another trend in computer science has been the decomposition of complex data into a collection of simple tables via the relational database model [5]. Data records are grouped into tables of records with a similar structure. Data records are extracted from one or more tables by selecting records with specific values in certain fields. Records from different tables are *joined* by matching records according to the values of specific fields. The results from a table join can be utilized on a record-by-record basis, or condensed into summary information.

The relational model provides dual benefits: a simplified user interface for interactive data exploration, and a manageable analytic model for programmers to use in accessing data. In particular, the Structured Query Language (SQL) has evolved as a flexible tool that can be applied to a wide variety environments, with compile-time or runtime optimization for efficient execution.

Using the tools of active networking, this paper applies the relational data model and SQL language to the data communication environment. The central metaphors in this paper may be summarized as follows:

> The state inherent in computer network nodes may be expressed as tabular data using a relational database model.
> Packet headers are analogous to relational database tables, and a packet is analogous to a joined set of records from multiple tables.
> Actions on packets may be expressed as changes to the relational records associated with the packets and/or node state.

2. Modeling node state as relational tables

The state of a computer network node is often remotely monitored through an information model called a Management Information Base (MIB). Although MIBs may be defined as heirarchical data objects, it is common to view and process them using a relational model as an interface. Extending this analogy further, you can also express changes in node state in a straightforward fashion as updates to records in relational tables.

3. Modeling packets as relational tables

Although it may not be apparent at first glance, collections of network communication packets are very similar to data in relational databases. Consider a set of packets P constructed from a fixed protocol stack, such as **ipv4/udp/anep**. Each packet in P consists of one instance each of an IPv4 header, a UDP header, and an ANEP (Active Network Encapsulation Protocol) header, stored adjacently in the packet. Each of the protocol headers consists of a set of fields. The set of IPv4 headers for all packets in P is analogous to the records of an **ipv4** relational database table, using the same set of fields; the set of UDP headers are analogous to the records of a **udp** table, and similarly for the set of ANEP headers.

Let us define each of the headers to have a virtual field, **packet_id**, containing a value that is unique to each packet in P; all headers in a single packet will have the same value for **packet_id**. Under this definition, the set of **ipv4/udp/anep** packets in P may be said in RDBMS terms to be a *prejoined* (also called *clustered*) physical storage representation for the logical records of the separate **ipv4**, **udp**, and **anep** tables, with **packet_id** as the *cluster key*.

Of course, there are differences in usage between the conventional relational database model and collections of packets. RDBMS (Relational Data Base Management System) usually operate on data that has been indexed for efficient access, while network packets are commonly acquired in real-time event streams (although it is not uncommon to collect packet traces and perform retrospective analyses). Many network protocols, such as IPv4, TCP, and ANEP, contain varying-length optional contents, the structure of which might best be *normalized* into multiple tables in an RDBMS model. However, as will be shown, it is possible to gain a considerable advantage from the relational database model of network packets without addressing these concerns in detail.

This completes the mapping between network packets and RDBMS records as stored in a typical RDBMS. By recasting network packets into the relational database model, we gain access to the tools and methodologies that have been developed in the last three decades for processing data in this form. In particular, we have access to the database query language SQL (Structured Query Language) [14] and tools based on it.

4. Applying the relational model to network monitoring

The Active Networks Query Language (ANQL) is an SQL-like application language. As a component of a network monitoring tool, it can be used to extract, summarize, and reformat information about packets, singly or in groups, in real-time event streams or in stored datasets. In this section we will first apply standard SQL to example network management problems. ANQL will introduce specialized syntax to reduce certain complexities of using SQL in typical protocol processing situations, and to make available further application-specific language features.

4.1. SQL examples

Suppose we wish to extract a trace of the source and destination IP addresses of every packet in a sequence of packets. Assume that the packet data was collected at some prior time, and is stored in relational tables as described in Section 3. In this example we need only a single table, **ipv4**, to hold IPv4 header data from each packet.[1] Fig. 1 shows an SQL query on this data (written as a database *view*). The IPv4 protocol header address fields are stored in database fields named **saddr** and **daddr** in the **ipv4** table. The **FROM** clause identifies this table as the data source for the SQL query, and the **SELECT** clause the data fields that are to be extracted for the output of the query.

Let us consider a more complex (yet not atypical) operation, such as decoding RIP [8] packets. For this example, we will use a RIP (Routing Information Protocol) implementation operating in an ASP EE (Active Signaling Protocol Execution Environment) [2] virtual network topology running on the ABone (Active Network Backbone) [1].[2] We want to extract the source and destination virtual addresses and the RIP command from each packet. Furthermore, we have to filter these packets out of a general packet stream that may contain many types of packets, only some of which are of interest to us. Fig. 2 shows one possible SQL command to do this.

```
CREATE VIEW ip_packets
AS
SELECT ipv4.saddr, ipv4.daddr
FROM ipv4
```

Fig. 1. SQL statement for extracting IP addresses.

```
CREATE VIEW rip_packets1
AS
SELECT vn.saddr, vn.daddr,
       decode(rip.command,
              1, "rip_request",
              2, "rip_response",
                 "rip_other") command_name
FROM ipv4, udp, anep, vn, vt, asp, rip
WHERE ipv4.protocol = 17
  AND  udp.packet_id = ipv4.packet_id
  AND  udp.dport     = 3322
  AND anep.packet_id = ipv4.packet_id
  AND anep.typeid    = 135
  AND   vn.packet_id = ipv4.packet_id
  AND   vt.packet_id = ipv4.packet_id
  AND   vt.dport     = 520
  AND  asp.packet_id = ipv4.packet_id
  AND  asp.aaname    = "rip"
  AND  rip.packet_id = ipv4.packet_id
```

Fig. 2. SQL statement for examining RIP packets.

The **SELECT** clause extracts the virtual source and destination addresses (**vn.saddr** and **vn.daddr**) and a text representation of the RIP command (using **decode**(...), which is a function for mapping data values that is found in some dialects of SQL). SQL has the expressive power to handle this example, but the large number of conjunctions in the **WHERE** clause is clumsy.

[1] In this paper, protocol data will be represented by a database table with the same name as the protocol, except for case.

[2] The ASP EE supports the protocols identified in the example as VN, VT, and ASP. The ABone supports the protocol identified as ANEP, as well as the constants used to match UDP port 3322 and ANEP TypeID 135.

5. ANQL examples

Fig. 3 shows an ANQL implementation of the example shown in Fig. 2. **CREATE ACTIVE FILTER** states that this particular ANQL statement is creating an Active Network packet filter (as opposed to a database view). Compared to SQL, the primary changes are that the **FROM** clause contains a protocol specification (see Appendix A) for particulars) and the **WHERE** clause no longer requires the many explicit comparisons on the **packet_id** field: the relational view of the data in the packet is implicitly pre-joined by ANQL. **USING NETIOD** specifies that the packets are to be acquired in real time through Netiod [1], a program that provides system-independent access to packet flows on Unix systems.

The ANQL **WHERE** clause contains the information that a NodeOS [7] channel specification carries in the address specification and optional demux specifications. Unlike the NodeOS positional notation, the **protocol.field** notation in ANQL is easily expandable and relatively self-documenting.

There are still some redundancies in ANQL. In Fig. 3, WHERE ipv4.protocol = 17 says that the protocol after IPv4 should be UDP. Although ANQL could have automatically added this constraint to the **WHERE** clause based on the contents of the **FROM** clause, ANQL takes the approach that all magic numbers that define protocol relationships should be shown explicitly; this is a user interface issue more than an essential property of the ANQL language itself.

The SQL database language, as implemented by commercial database vendors, can perform data transformations in the **SELECT** and **WHERE** clauses. In addition to the usual numeric and logical operations, SQL implementations often support string manipulation, date/time calculation, table-based data value remapping, and other non-numeric operations.

The initial ANQL implementation supports many of the data transformation capabilities that are typical of SQL, and adds specialized functions that are appropriate to the area of Active Networking. Fig. 4 extends the **rip_packets2** filter by calling the **vnetToHost**(...) function to convert a virtual network address into a host name. The **AS** syntax in the **SELECT** clause is used to rename the computed output fields. The result is a tuple with fields named (**node**, **node2**, **property**, **status**), which meets the input requirements of the network packet visualizer [10] for which this script was used to filter packets in real time.

SQL provides a **GROUP BY** clause to direct the computation of summary statistics. In ANQL, it can be used to summarize over time or over the attributes of the packets. For example, Fig. 5 extracts the bandwidth consumed by each RIP command type, in octets per second averaged over 10-min intervals. The **interval**(...) function shown here is an example of an ANQL extension to SQL that was created to better adapt the language to the packet processing domain, in this case by simplifying the manipulation of time intervals.

```
CREATE ACTIVE FILTER rip_packets2
AS
SELECT vn.saddr, vn.daddr,
       decode(rip.command,
              1, "rip_request",
              2, "rip_response",
                 "rip_other") as status
FROM "if/ipv4/udp/anep/vn/vt/asp/rip"
WHERE ipv4.protocol = 17
  AND  udp.dport  = 3322
  AND anep.typeid = 135
  AND   vt.dport  = 520
  AND  asp.aaname = "rip"
USING NETIOD
```

Fig. 3. ANQL statement for examining RIP packets.

```
CREATE ACTIVE FILTER rip_packets3
AS
SELECT vnetToHost(vn.saddr) AS node,
       vnetToHost(vn.daddr) AS node2,
              "rip-packet!" AS property,
       decode(rip.command,
              "1", "rip_request",
              "2", "rip_response",
                   "rip_other") as status
FROM "if/ipv4/udp/anep/vn/vt/asp/rip"
WHERE ipv4.protocol = 17
  AND  udp.dport = 3322
  AND anep.typeid = 135
  AND   vt.dport  = 520
  AND  asp.aaname = "rip"
USING NETIOD
```

Fig. 4. ANQL for extracting and reformatting RIP data.

```
CREATE ACTIVE FILTER rip_bandwidth
AS
SELECT rip.command, sum(ipv4.length)/(10*60) AS bandwidth
FROM "if/ipv4/udp/anep/vn/vt/asp/rip"
WHERE ipv4.protocol = 17
  AND  udp.dport = 3322
  AND anep.typeid = 135
  AND   vt.dport  = 520
  AND  asp.aaname = "rip"
GROUP BY rip.command, interval(sysdate, 10, "MI")
USING NETIOD
```

Fig. 5. ANQL for calculating RIP bandwidth.

6. Using node state in network monitoring

The previous examples selected candidate packets solely based on the contents of each packet. This model can be extended by providing access to relational tables that represent the state of the node on which the packet is being evaluated and processed. For example, consider a simple **linkTrust** table that indicates whether each interface has been configured as trusted or untrusted. Fig. 6 demonstrates a barebones filter that extracts the source and destination addresses of IPv4 packets that arrive from an untrusted link. The **FROM** clause uses a combination of a protocol specification for accessing packet data and conventional SQL **schema.table AS alias** syntax for accessing node state information.

Fig. 7 detects IP packets that arrive from the wrong interface, as indicated by a table of legal IP prefixes per interface. In contrast, implementing the entire IP prefix lookup as a specialized function, as in Fig. 8, is notationally more concise and may be easier to implement efficiently.

Other categories of node state that may be of interest to active packet filters include link error

```
CREATE ACTIVE FILTER input_from_untrusted_link
AS
SELECT if.interface_number, ipv4.saddr, iv4.daddr
FROM "if/ipv4", node.link_trust AS lt
WHERE lt.interface_number = if.interface_number
  AND lt.is_trusted = "N"
USING NETIOD
```

Fig. 6. ANQL for intercepting IP packets from untrusted links.

```
CREATE ACTIVE FILTER input_from_wrong_link
AS
SELECT if.interface_number, ipv4.saddr, iv4.daddr
FROM "if/ipv4"
WHERE NOT EXISTS (
      SELECT 'X'
        FROM node.ip_prefixes AS ipp
       WHERE ipp.interface_number = if.interface_number
         AND ipAddressPrefixedBy(ipv4.saddr, ipp.prefix)
)
USING NETIOD
```

Fig. 7. ANQL for intercepting IP packets from wrong links.

```
CREATE ACTIVE FILTER input_from_wrong_link2
AS
SELECT if.interface_number, ipv4.saddr, iv4.daddr
FROM "if/ipv4"
WHERE NOT correctIPPrefix(if.interface_number, ipv4.saddr)
USING NETIOD
```

Fig. 8. Simplified ANQL for intercepting IP packets from wrong links.

rate, link saturation, node CPU utilization. More generally, any node state that is available for network monitoring using a conventional MIB can be made available to an active packet filter through a SQL schema.

7. Classifying and annotating packets

ANQL can be used to classify packets for further analysis by feeding the output from one packet filter into the input of another packet filter. This might be done to extract shared code from a set of related ANQL filters as an efficiency trade-off, or to isolate configuration parameters to a single ANQL statement. Fig. 9 creates separate **rip_request** and **rip_response** tables from a shared RIP classifier. Another option is to annotate a packet with derived values without anticipating which other packet fields will be needed by further packet filters (see Fig. 10). In either case, the ANQL implementation automatically analyzes the **FROM** dependencies to execute the active filters in the correct order.

8. Network monitoring and control applications

The initial applications supported by ANQL are in the control or management planes in a network. In these applications, ANQL scripts function as commands to management application programs (which might be implemented as active applications or as non-active ones), and are not strictly speaking active packets. On the other hand, ANQL scripts

```
CREATE ACTIVE FILTER rip_classifier
AS
SELECT vn.saddr AS saddr, vn.daddr AS daddr, rip.command AS command
FROM "if/ipv4/udp/anep/vn/vt/asp/rip"
WHERE ipv4.protocol = 17
  AND  udp.dport = 3322
  AND anep.typeid = 135
  AND   vt.dport  = 520
  AND  asp.aaname = "rip"
USING NETIOD

CREATE ACTIVE FILTER rip_request
AS
SELECT rc.saddr, rc.daddr
FROM rip_classifier AS rc
WHERE rc.command = "1"

CREATE ACTIVE FILTER rip_response
AS
SELECT rc.saddr, rc.daddr
FROM rip_classifier AS rc
WHERE rc.command = "2"
```

Fig. 9. Chaining active filters.

```
CREATE ACTIVE FILTER rip_annotation
AS
SELECT "Y" AS is_rip
FROM "if/ipv4/udp/anep/vn/vt/asp/rip"
WHERE ipv4.protocol = 17
  AND  udp.dport = 3322
  AND anep.typeid = 135
  AND   vt.dport  = 520
  AND  asp.aaname = "rip"
USING NETIOD

CREATE ACTIVE FILTER rip_request
AS
SELECT vn.saddr, vn.daddr
FROM "if/ipv4/udp/anep/vn/vt/asp/rip", rip_annotation
WHERE rip_annotation.is_rip = "Y"
  AND rip.command = "1"

CREATE ACTIVE FILTER rip_response
AS
SELECT vn.saddr, vn.daddr
FROM "if/ipv4/udp/anep/vn/vt/asp/rip", rip_annotation
WHERE rip_annotation.is_rip = "Y"
  AND rip.command = "2"
```

Fig. 10. Sharing a packet annotation.

may be used to create distributed sessions for management reporting (such as by creating a tree of ANQL scripts passing summary information towards a common root), and in this application ANQL scripts may operate as active packets.

8.1. Filtering packets

The initial uses for ANQL have been to filter packets from remote packet intercept points. A single ANQL processor runs on a central system, operating on remotely gathered data or even on stored packet traces; the examples in Section 5 illustrate this capability. An ANQL-based filter may also be distributed to packet collection points in an Active Network topology, with little or no change in the filter itself. Thus, the ANQL-based filters can scale beyond the processing limits of a centralized filter.

8.2. Summary statistics

ANQL scripts can also be used to collect summary statistics on packet flows in an Active Network. In a small network, summary statistics can be computed by a single, central node that collects intercepted packet streams from all other nodes in the topology. In a larger network, the ANQL script can be distributed to a selection of nodes in the topology, with little change in the ANQL script. Beyond that, ANQL scripts can be distributed to compute summary statistics in a heirarchical computation tree, with little change to the ANQL script. This provides a high degree of scalability.

ANQL, as a dialect of SQL, provides **MIN**, **MAX**, **SUM**, **AVG**, **STDDEV**, and other descriptive statistics in the language. Since many summary statistics of packet traffic are supported by ANQL itself, that code does not need to be written into Active Applications (AAs) that require the summary data. This greatly simplifies the implementation and maintenance of these AAs.

8.3. Triggering actions on nodes

ANQL scripts can be used to trigger actions in an Active Network. Fig. 11 has an active trigger that

```
CREATE ACTIVE TRIGGER restart_neighbor
AS
SELECT "restart "||vn.saddr AS action
FROM "if/ipv4/udp/anep/vn/vt/asp/rip"
WHERE ipv4.protocol = 17
  AND  udp.dport = 3322
  AND anep.typeid = 135
  AND   vt.dport  = 520
  AND  asp.aaname = "rip"
  AND  rip.command != "1"
  AND  rip.command != "2"
GROUP BY vn.saddr, interval(sysdate, 10, "MI")
HAVING count(*) > 3
USING NETIOD
```

Fig. 11. ANQL for restarting failing neighbors.

uses the SQL ''group by'' and ''having'' clauses (in their ANQL incarnations) to issue restart commands for neighboring nodes that have sent more than 3 erroneous RIP packets in a 10-min interval. The restart action is created as a text string containing the word ''restart'' and the virtual network address of the failed node; presumably this is a command to the application using ANQL.

It would not be difficult to use ANQL to generate operating system commands (shell commands) for execution on local or remote nodes. Report-writing scripts for early versions of Oracle's SQL tools often operated in an analogous fashion.

9. ANQL and active network architecture

ANQL can be integrated into an active network architecture in several ways. Placing ANQL above a low-level packet filter such as Netiod [1] (see Section 12), provides a highly-expressive mechanism for filtering packets for delivery to an Execution Environment (EE) or Active Application (AA). This is the basis of the examples in the preceding sections.

The ANQL **FROM** clause may contain a NodeOS-like protocol specification. The ANQL **WHERE** clause may contain the information that is carried in the NodeOS address specification and optional demux specification(s). Unlike the positional notation used in the NodeOS address specification, ANQL's **protocol.field** notation is easily expandable and relatively self-documenting. As a result, it is expected that ANQL packet filters will be easier to program and maintain than equivalent functions written as NodeOS channel specifications. Thus, ANQL could be used to improve program quality by replacing the NodeOS channel specification in an EE.

In addition to packet filtering and network monitoring, suitably configured ANQL implementations can be used to make decisions that control the processing of individual packets. ANQL could be used as a capsule language (see Section 10) in a specialized Execution Environment. Many of the decision-making functions of an EE (such as access control and topology monitoring) could be implemented in the EE itself using ANQL statements, in addition to the use of ANQL in capsule programs.

ANQL could be hosted on top of the ANTS (Active Network Transfer System) or ASP EEs by writing an ANQL interpreter as an AA for those systems, or by encapsulating or translating ANQL into Java for execution as AAs. This would allow the exploration of ANQL as a high-level capsule language, without requiring the creation of an entire new EE.

10. Capsule applications of ANQL

Capsule languages are languages used to write Active Network programs that are carried directly in active packets; one example is [11]. Due to packet size limitations in typical computer networks, capsule languages face a difficult tradeoff between brevity and functionality. The focus of a capsule language, operating in the *data plane* of an active network, tends to center on the selection of node resources to match the processing requirements of the packet.

ANQL can be used as an active networking capsule language. Instead of filtering and manipulating the contents of one or more packets into a monitoring channel, ANQL can support tasks such as selecting among protocol processing modules in a node or selecting from a set of nodes for packet forwarding, using a combination of the available set of resources at each node traversed by the capsule and the data carried in the packet itself. Fig. 12 has a simple example that illustrates this approach. The sample program selects the node with the shortest queue as the next node for the current packet. Ties are broken by selecting the node with the lowest address, due to the **ORDER BY n.addr** clause. The **CREATE ACTIVE FILTER simple.next_node** clause identifies this as the **next_node** annotation in the capsule named **simple**. An ANQL execution environment (ANQLEE) would accept packets that are

```
CREATE ACTIVE FILTER simple.next_node
AS
SELECT n.addr AS next_node
FROM node.neighbor_nodes AS n
WHERE n.queue_length in (
    SELECT min(n2.queue_length)
    FROM node.neighbor_nodes AS n2
)
ORDER BY n.addr
```

Fig. 12. SQL for active packet routing.

labeled with ANQL capsules, and process them according to the capsule-specific ANQL statements.

As described in Section 13, ANQL could be augmented with a procedural extension. The result, a computationally complete language, could be compiled into byte code and used as an Active Network capsule language: the source code would be high-level and self-documenting, while the compiled byte code could easily be as at least as compact as Java's byte code.

11. Security issues

A frequent concern raised about active networks is the potential for unbounded security problems associated with user-directed dissemination of programs for execution on network nodes. There are concerns that these programs may access inappropriate data on the nodes, compromising the security of other users, and may take over node execution, compromising the network itself. Much research effort in projects such as NodeOS and ASP has been spent developing mechanisms that control the potential access from user-initiated programs to the underlying node architecture.

Whether used for packet filtering, EE implementation, or as a capsule language, ANQL provides levels of implementation control that can greatly improve the security of an Active Network.

ANQL can provide access to a variety of per-packet and node state data, with all access passing through the ANQL plug-in data sources. Compared to writing capsule programs in C or even Java, this architecture greatly reduces the possibility of inadvertent data exposure. Although ANQL statements can be very expressive, especially when augmented with a procedural extension (see below), ANQL's mediated access provides a strict control on the data that may be accessed. If ANQL is used to control decision-making activities, such as next-hop link selection in an ANQL capsule, the high-level ANQL architecture, with mediated data outputs, can be used to prevent inadvertant or malevolent changes to the active node's state.

Because of their high-level, application-oriented expression, ANQL statements could be analyzed for validity, data sources, and data outputs before execution. This should be much easier to accomplish than analyzing a capsule program transmitted as, e.g., Java byte code. After analysis, ANQL statements can be translated into byte code, C, Java, or machine-level code for efficient execution, retaining confidence that the resulting programs are secure. The ANQL interpreter, translator, and plugins themselves can be analyzed and audited to ensure that the desired data access limitations are enforced.

The ANQL implementation's plugin architecture can be used to further limit access to certain data based on AA or capsule authorizations, such as can be provided through code signatures. Used in an AA or capsule setting, ANQL's data sources can be individually tailored to the authorizations of the user or process that signed the AA or capsule. Thus, ANQL capsule programs from one source may be permitted access and control over packet routing mechanisms, while ANQL capsule programs from another source may be limited to observing and recording specific portions of node state.

Non-capsule applications of ANQL can also improve active network security. For example, ANQL can be used as a replacement for the NodeOS channel specification. Because ANQL provides named access to protocol fields used for filtering, rather than positional access, it is expected that programmers will make fewer errors in coding their channel specifications. Typographic errors, which can be a source of accidental data exposure, are more likely to be caught during execution when fields are named rather than positional. If desired, a precompiler pass could be used to catch errors in ANQL embedded in another language, such as Java, prior to execution. The ANQL implementation's plugin architecture could be used by an EE to facilitate restricting certain channel filter capabilities to individual AAs based on validated authorizations.

12. Implementation considerations

ANQL is an application-specific language. It expresses application-specific requests more

compactly than a more general purpose language (such as Java byte code). Thus, for many purposes ANQL commands can be included directly in packets, consuming less bandwidth than an equivalent program written in Java or C.

ANQL is currently implemented using a general expression interpreter [9] written in Java. This implementation decision provided a high degree of functionality at the cost of runtime efficiency. The design of ANQL does not preclude compilation to Java or C, or even to machine code for use on specialized network protocol processors. The ANQL language (excluding the possible procedural extensions discussed elsewhere in this paper) is non-procedural; this quality should make it easier to compile ANQL to platform-specific code.

There are certain optimizations that could be applied to the present implementation of ANQL. For example, when processing protocol headers, all possible protocol fields (**ipv4.saddr**, **ipv4.daddr**, etc.) are extracted from the packet and saved in a Java util.Hashtable. It should be possible to analyze the protocol fields used in the ANQL expressions and extract only those fields that are specifically needed.

ANQL uses Netiod [1], an operating-system neutral packet filter interface, to implement portions of its packet acquisition and filtering process. ANQL could be implemented on top of the BSD Packet Filter [12] or on top of a more system-specific packet filtering mechanism, such as IPCHAINS or IPTABLES on Linux. In all of these cases, ANQL extends the functions of the lower-level packet filter with ANQL's SQL-based syntax and semantics. ANQL acts as a higher-level filtering layer, and uses the available lower-level filtering functions to improve operating efficiency.

In addition to packet-based data, ANQL mediates controlled access to other sources, such as configuration data, dynamic node state, MIB-based data, etc., through plug-in modules in the ANQL implementation. Each data source appears as one or more named relational tables that may appear in an ANQL **FROM** clause.

13. Further research

The ANQL language continues to grow to meet the requirements of network control and management, and of Active Networking in particular. Additional functions are being written for use in ANQL expressions, and the range of protocols that can be parsed by ANQL continues to expand.

ANQL's syntax is similar to Oracle's implementation of SQL, which is well known and relatively accessible. There are other languages related to ANSI SQL with interesting extensions, such as time comparisons, that might be desirable in the Active Networks domain. It could be useful to incorporate these features into ANQL.

Oracle Corp. has created a procedural language extension to SQL, PL/SQL. Microsoft's SQL Server also has procedural extensions. A procedural extension is feasible for ANQL; such an extension could be used as a portable, computationally complete active packet language. This approach represents an interesting tradeoff between semantic expressiveness, representational compactness, and implementation complexity, compared to prior efforts[11]. If ANQL were hosted on top of ASP or ANTS, it would be easy to use Java as a procedural extension language.

A non-procedural language, such as ANQL, may also have certain advantages when analyzing the safety of statements in the language. It would be interesting to pursue the safety and security properties of ANQL.

ANQL could replace the NodeOS channel specification in some programs. The translation from ANQL to lower-level channel mechanisms is a one-time operation, which may not be significant when amortized over the lifetime of a packet flow. If necessary, the translation could be cached for reuse at runtime (if the same channel specification is opened multiple times in the lifetime of an EE) or precompiled (in the style of commercial SQL precompilers) into the source code of an active application.

14. Related work

NNStat [3,4] provides a low-level packet filter, a remote packet collection facility, and several higher-level data analysis capabilities. ANQL and NNStat use similar **protocol.field** naming conventions, but NNStat is much more concerned with the lower-level details of the packet filter implementation than is ANQL, which focuses on higher-level issues.

The AT&T Labs Gigascope project [6] also applies the relational database model to monitoring data network packets, and incorporates a flexible SQL-like query language. Gigascope compiles monitoring requests and executes them in a highly efficient manner. ANQL attempts to address a somewhat broader research domain in an Active

Networks environment, including packet forwarding and filtering.

15. Summary and conclusions

The Active Networks Query Language applies the expressive power of SQL to the domain of packet network control and management. This increase in expressive power makes ANQL easier to use than NodeOS channel specifications in many applications; furthermore, many common operations, such as computing summary statistics about packet traffic, can be expressed directly in ANQL rather than requiring custom code in each application that needs code. The initial implementation of ANQL is an interpreter written in Java, but ANQL could alternatively be compiled directly into Java, C, or machine code for efficient execution.

Appendix A. The ANQL language

The initial implementation of ANQL closely resembles SQL.

```
CREATE ACTIVE [FILTER | TRIGGER]
<name>
AS
SELECT <selectExpr> [AS <fieldName>]
[, ...]
FROM <protocolSpec> | <filter_name>
[, <filter_name>...]
[WHERE <whereExpr>]
[GROUP BY <groupExpr>]
[HAVING <havingExpr>]
[ORDER BY <orderExpr>]
[USING NETIOD]
```

The expressions may contain logical operators, arithmetic operators, comparisons, functions, etc. In general, these behave as they would in SQL. ANQL's functionality can be easily increased because the ANQL interpreter can be dynamically extended at runtime by loading Java code to implement new operators and functions.

The **WHERE** clause selects records before grouping and the **HAVING** clause selects record groups after grouping, as in SQL. ANQL's **FROM** clause uses a protocol specification that is similar to the NodeOS channel specification, such as:

```
FROM "if/ipv4/udp/anep/vn/vt/asp/
rip"
```

The protocol names are separated by slashes. Each protocol, when matched against an incoming packet, makes certain data fields available for use in expressions in the ANQL statement. To repeat a protocol, such as in IP/IP tunneling, a colon-separated suffix can be attached to each protocol name to disambiguate the protocol layers in ANQL expressions; this suffix mechanism is similar to the table name alias feature of some SQL implementations. Example:

```
SELECT ipv4:2.saddr
FROM "if/ipv4:1/ipv4:2/udp/anep/vn/
vt/asp/rip"
```

Here is another example, showing IPv4 tunneling within a UDP envelope. In this case, only the outer IPv4 and UDP headers have been given a special suffix, and the **SELECT** extracts the inner IPv4 header:

```
SELECT ipv4.saddr
FROM "if/ipv4:outer/udp:outer/ipv4/
udp/anep/vn/vt/asp/rip"
```

In addition to the protocols and fields mentioned in the **FROM** clause of an ANQL statement, certain special fields may be available in an ANQL expression. For example, when using Netiod to intercept packets, **netiod.raddr** may be the IP address of the Netiod instance that intercepted the packet, and **netiod.timestamp** may be the Netiod-supplied timestamp for when the packet was intercepted.

As of this writing, ANQL parsers have been implemented for the following network protocols:

```
General: IPv4, RDP, RIP, TCP, UDP
Active Networking: ANEP, Netiod
ASP EE: AASpec, UI, VNP, VNS, VTP, VTS
```

As an example, the IPv4 fields that are available for use in ANQL expressions are:

```
ipv4.length, ipv4.df, ipv4.mf, ipv4.ttl,
ipv4.tos, ipv4.protocol,
ipv4.saddr, ipv4.daddr
```

Appendix B. The ANQL implementation in the ABoneMonitor

ANQL is implemented as a Java package that may be extended and used in a variety of environ-

ments. The initial application for ANQL was the ABoneMonitor, an automated utility for testing and logging the connectivity of ABone nodes. Test results may be displayed on an animated GUI (Graphical User Interface) in real-time, or by replaying a log of recorded events. During real-time monitoring, ANQL-driven network monitors may be added or removed dynamically.

An ANQL monitor may be added by typing an ANQL statement into an Web form, or be recalling an existing ANQL statement and editing it in the Web form. When submitted to the ABoneMonitor, the ANQL statement is parsed into component clauses (SELECT, FROM, WHERE, etc.). The data requirements of the ANQL statement are analyzed, and data sources, such as NetIOD channels, are opened as needed on selected network monitoring nodes.

When possible, the ANQL monitor pushes packet filtering operations to the low-level packet data source. If NetIOD is used as a data source, the ANQL statement is matched against acceptable NetIOD filter patterns, such as selecting packets from specific source or destination addresses from an "if/ipv4" filter specification. An appropriate NetIOD request is built as an output of this filter optimization analysis.

The ABoneMonitor monitors the status of its NetIOD filters. If a connection to NetIOD is lost (because the NetIOD process crashed, the node on which it was running died, or network links made the node unavailable), the ABoneMonitor will attempt to reestablish communications with NetIOD on the target node. When communications are reestablished, the ABoneMonitor scans its list of active ANQL statements and rebuilds the necessary NetIOD channels and filters.

When a packet is received from a data source, it is processed by one or more ANQL statements that are monitoring packets from the data source from which the packet arrived. Each ANQL statement runs the packet against to a chain of protocol analyzers, which check, when possible, whether the next protocol in the packet is the protocol required by the ANQL statement. Each protocol analyzer may extract fields from its protocol header in the incoming packet, and enter these fields in a symbol table that is specific to the packet and ANQL statement being processed.

After the packet's symbol table has been built for an ANQL statement, it is passed to an expression interpreter to implement filters expressed in the WHERE clause of the ANQL statement. The expression interpreter is an extensible stack-based package that was implemented for the ABoneShell, an interactive shell-like language that can be used to monitor the ABone by querying monitoring ABone peers (NetIOD, ANetD, the ASP EE, etc.).

If the packet passes the ANQL statement's WHERE clause, then the ABoneShell interpreter is called to evaluate the ANQL statement's SELECT clause. This results in a new symbol table.

If there is no GROUP BY clause, the results of the SELECT clause are converted into an ABoneMonitor event entry and fed into the currently active event log stream. If a GROUP BY clause was present, grouped data is accumulated in a symbol table associated with the ANQL statement; this grouped data may be fed into the ABoneMonitor event log for display of cumulative results.

References

[1] Steven Berson, Steven Dawson, Robert Braden, Evolution of an active network testbed, in: DARPA Active Networks Conference & Exposition, May 2002, , pp. 446–465.

[2] Bob Braden, Alberto Cerpa, Ted Faber, Bob Lindell, Grahap Phillips, Jeff Kahn, ASP EE: an active execution environment for network control protocols, 1999. Available from: <http://www.isi.edu/active-signal/ARP>.

[3] Robert T. Braden, A Packet Monitoring Program, USC/Information Sciences Institute, 1990.

[4] Robert T. Braden, Annette L. DeSchon, NNStat: Internet Statistics Collection Package: Introduction and User Guide, USC/Information Sciences Institute, 1988.

[5] E.F. Codd, A relational model of data for large shared data banks, Communications of the ACM 13 (6) (1970) 377–387.

[6] Chuck Cranor, Yuan Gao, Theodore Johnson, Vlaidslav Shkapenyuk, Oliver Spatscheck, GIGASCOPE: High performance network monitoring with an SQL interface, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2002.

[7] AN Node OS Working Group, NodeOS Interface Specification, January 2000. Available from: <http://www.cs.princeton.edu/nsg/papers/nodeos.ps>.

[8] G. Malkin, RIP Version 2, RFC 2453, November 1998.

[9] Craig Milo Rogers, The ABoneShell. Available from: <http://www.isi.edu/abone/ABoneShell.html>.

[10] Craig Milo Rogers, ABoneMonitor Packet Visualizer Demo, DANCE 2002, San Francisco, CA, May 2002.

[11] B. Schwartz, W. Zhou, A.W. Jackson, et al., Smart Packets for Active Networks, Technical Report, BBN Technologies, January, 1998.

[12] Steven McCanne, Van Jacobson, The BSD Packet Fitler: A new architecture for user-level packet capture, in: Proceedings of the Winter 1993 USENIX Conference, January 1993, pp. 259–270.

[13] D.L. Tennenhouse, D.J. Wetherall, Towards an active network architecture, Computer Communications Review, 1996. Available from: <http://www.tns.lcs.mit.edu/publications/ccr96.html>.
[14] ANSI X3.135, Database Language SQL, 1992.
[15] J. Zander, R. Forchheimer, SOFTNET – An approach to high level packet communications, in: Proceedings of the AMRAD Conference, 1983.

Craig Milo Rogers is a member of the technical staff of USC/Information Sciences Institute since 1980. He was a programmer/analyst at UCLA from 1971 to 1980. He learned to program on the AN/FSQ-32 at System Development Corp. in 1968. His professional interests include network protocols, relational databases, and user interfaces for data visualization.