ANQL – An Active Networks Query Language

Craig Milo Rogers

USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA, USA
rogers@isi.edu

Abstract. This paper discusses parallels between network communication packets, when processed in bulk, and relational database records. It introduces a new application-specific language, ANQL (Active Networks Query Language), that exploits a database metaphor for packet processing. ANQL has been demonstrated in Active Network control and management plane activities, although it may also be used in many other networking applications. In active networks, ANQL is primarilly intended as a tool or adjunct for use by Active Applications, and by control and management code. Environments are discussed in which ANQL or related languages might be utilized as full-fledged active packet languages in themselves. ANQL is applicable to both event-driven and background processing activities, and may be used in a single, centralized data collection and analysis process, or, with little change, in distributed implementations of packet analysis activities.

1 Introduction

There are many tasks that need to be performed in the control and management planes of a data communication network. Some tasks include extracting or acting on packets that meet certain conditions; representative tasks include:

- checking for faults
- detecting intrusions
- monitoring content for viruses
- identifying packets for which special routing or services are applied

Other tasks may require taking actions based on aggregate values calculated on groups of packets. These tasks may include:

- sending an alarm when a sequence of packets indicates that an attack is in progress
- detecting when excessive traffic of a particular class is monopolizing network resources
- reporting bandwidth utilization statistics

Problems of this sort have been present since computer networks first became available, and many solutions have been implemented in the past. The advent of Active Networking [14] [12] gives us a new set of tools that we may apply to the control and management of computer networks, providing opportunities to create better solutions to long-existing network management problems.

2 Comparing Packets to RDBMS Data

The central metaphor in this paper may be summarized as follows:

```
Computer network packets are analogous to table joins in a
relational database management system (RDBMS).
```

In 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.

Although it may not be apparent at first glance, collections of network communication packets are very similar to relational database data. 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 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 systems 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) [13] and tools based on it.

3 Applying the RDBMS Model

The Active Networks Query Language (ANQL) is an SQL-like application language that 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 be introduced to reduce certain complexities of using SQL typical protocol processing situations, and to make available further application-specific language features.

3.1 SQL Examples

Let us apply the RDBMS model to a simple problem in network management. 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 collected at some prior time, and is stored in RDBMS tables as described in Sect. 2. In this example we need only a single table, **ipv4**, to hold IPv4 header data from each packet [1]. Figure 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**, which are accessed using the SQL notation **ipv4.saddr** and **ipv4.daddr** to clarify the data source involved.

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

Fig. 1. SQL statement for extracting IP addresses.

Let's consider a more complex (yet not atypical) operation, such as decoding RIP [7] packets. For this example, we will use a RIP implementation operating in an ASP EE [2] virtual network topology running on the ABone [1] [2]. We want to extract the source and destination virtual addresses and the RIP command

[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.

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. Figure 2 shows one possible SQL command to do this.

```
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.

4 ANQL Examples

Figure 3 shows the same example, using ANQL. Compared to SQL, the primary changes are that the **FROM** clause contains a protocol specification (see Appendix A) for particulars) and the **WHERE** clause is devoid of the implicit comparisons on **packet_id**. **CREATE ACTIVE FILTER** states that this particular ANQL statement is creating an Active Network packet filter (as opposed to a database view). **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 [6] channel specification carries in the address specification and optional demux

```
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.

specifications. Unlike the NodeOSpositional notation, the **protocol.field** notation in ANQL is easily expandable and relatively self-documenting.

There are still some redundancies in ANQL. In Figure 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 numerical 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. Figure 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 [9] 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 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 minute intervals. The **interval(...)** function shown here is an example of an ANQL extension to

```
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.

SQL that was created to better adapt the language to the packet processeing domain, in this case by simplifying the manipulation of time intervals.

```
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.

5 Applications of ANQL

ANQL is an application-specific language. As such, it can express application-specific requests more compactly than a more general purpose language (such as general Java byte code). Thus, for many purposes ANQL commands can be included in packets without sacrificing a lot of bandwidth.

The primary applications envisioned for ANQL are in the control or manangement planes in a network. In these applications, ANQL scripts function as commands to control or management application programs (which might be implemented as active applications or as inactive ones), and are not strictly speaking active packets. On the other hand, ANQL scripts 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.

5.1 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 Sect. refsec:anqlExamples 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.

5.2 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 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.

5.3 Triggering Actions

ANQL scripts can be used to trigger actions in an Active Network. Figure 6 has an active trigger that 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-minute 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. The security implications of this technique are outside the scope of this paper.

```
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. 6. ANQL for restarting failing neighbors.

6 Relation to the NodeOS Channel Spec

The ANQL **FROM** clause contains a NodeOS-like protocol specification. The ANQL **WHERE** clause contains 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 equivalents programs written directly in NodeOS channel specifications.

7 Implementation and Efficiency

ANQL is currently implemented using a general expression interpreter [8] 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 the fields that are specifically needed.

8 Capsule Applications of SQL-like Languages

Capsule languages are languages used to write Active Network programs that are carried directly in active packets; one example is [10]. 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, as described here, focuses on the extraction of data from individual packets or groups of packets; it is not really suitable for use as a capsule language.

It is possible to envision a role for an alternative SQL-like language as a pure capsule language. Instead of filtering and manipulating the contents of one or more packets, the language would 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. Figure 7 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):

```
SELECT n.addr next_node
FROM neighbor_nodes n
WHERE n.queue_length in (
    SELECT min(n2.queue_length)
    FROM neighbor_nodes n2
)
ORDER BY n.addr
```

Fig. 7. SQL for active packet routing.

As described in Sect. 9, the SQL-like capsule language could be augmented with a procedural extension. The result, a computationally complete language, could be compiled into byte codes and used as an Active Network capsule language: the source code would be high-level and self-documenting, while the compiled byte codes could easily be as compact as Java's, if not more so.

9 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 range of protocols that can be parsed by ANQL continues to expand.

ANQL 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 features, such as time comparisons extensions, that might be desirable in the Active Networks domain. It could be useful to incorporat these features into ANQL.

Oracle Corp. has created a procedural language extension to SQL, PL/SQL. There is also an ANSI SQL procedural extension for SQL. A procedural extension is feasible for ANQL; such an extension could be used as a portable, computationally complete active packet language.

We would like to investigate a SQL-based capsule language, as described in Sect. 8. This approach represents an interteting tradeoff between semantic expresiveness, representational compactness, and implementation complexity, compared to prior efforts[10].

A nonprocedural 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.

10 Related Work

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 [11] or on top of a more system-specific packet filtering mechanism, such as IPCHAINS or IPTABLES on Linux. In all of these cases, ANQL would extend the functions of the lower-level packet filter with ANQL's SQL-based syntax and semantics.

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.

11 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 them. The initial implementation of ANQL is interpretive, but ANQL could alternatively be compiled directly into Java, C, or machine code for efficient execution.

References

[1] Steven Berson, Steven Dawson, and Robert Braden. Evolution of an Active Network Testbed. In *DARPA Active Networks Conference & Exposition*, pages 446–465, May 2002.

[2] Bob Braden, Alberto Cerpa, Ted Faber, Bob Lindell, Graham Phillips, and Jeff Kann. ASP EE: An Active Execution Environment for Network Control Protocols. `http://www.isi.edu/active-signal/ARP`, 1999.

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

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

[5] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[6] AN Node OS Working Group. NodeOS Interface Specification. `http://www.cs.princeton.edu/nsg/papers/nodeos.ps`, January 2000.

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

[8] Craig Milo Rogers. The ABoneShell. `http://www.isi.edu/abone/ABoneShell.html`.

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

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

[11] Van Jacobson Steven McCanne. The BSD Packet Fitler: A New Architecture for User-level Packet Capture. In *Proceedings of the Winter 1993 USENIX Conference*, pages 259–270, January 1993.

[12] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. In *http://www.tns.lcs.mit.edu/publications/ccr96.html*, 1996.

[13] X3.135. Database Language SQL. Technical report, ANSI, 1992.

[14] J. Zander and R. Forchheimer. SOFTNET - An Approach to High Level Packet Communications. In *Proceedings of the AMRAD Conference*, 1983.

A The ANQL Language

The initial implementation of ANQL closely resembles SQL.

```
CREATE ACTIVE [FILTER | TRIGGER] <name>
AS
SELECT <selectExpr> [AS <fieldName>] [, ...]
FROM <protocolSpec>
[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 disabiguate 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, TCP, UDP
Active Networking: ANEP, Netiod
ASP EE: AASpec, UI, VN, VT, 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
```