

# WHITE PAPER

## Network Query Language

A scripting language for rapid development of agents, bots, and other connected applications

This article introduces a new scripting language, Network Query Language. NQL was created with the new wave of software applications in mind: intelligent agents, bots, spiders, middleware, and web applications—which we'll collectively refer to as *connected applications*. In this article, we'll first examine the roots and justification for NQL, and then present the language itself along with some examples.

### I. New times bring new needs

If you're going to introduce a new programming language, you'd better have some pretty good reasons for doing so. NQL was created out of necessity, after repeated disappointment with existing programming languages. It's not that VB, C++, Java, or Perl are flawed in some way; they're all fine languages, each with valuable features and proud histories. It's merely that our focus has shifted. Software development has a different emphasis today than it did 5 years ago. The Internet and the corporate network are now key ingredients in solutions, and the web's influence has changed our thinking about how we approach many tasks. For the most part, programming languages haven't adapted to this change in direction.

There are four essential ingredients in developing connected applications: communications, conversion, automation, and intelligent behavior. Surprisingly, you'll find little or no direct support for these capabilities in today's programming languages. You almost always have to add on support, in the form of modules, classes, or components in order to get the job done. Even after doing so, code often ends up being overly long and complex. There's also an integration problem: since add-ons are generally ignorant of each other's existence and nature, they need a lot of help from the programmer in order to work cooperatively. Clearly, there is a problem here: most languages in use today lack the right building blocks and the right granularity for today's development needs.

### II. Learning a lesson from the past

Fortunately, we've seen this exact suite of problems in the past, and we can learn a lesson from one of the great success stories of the computer industry: SQL. During the '70s, the tremendous focus on databases brought to light weaknesses in the programming languages of the day: while adequate for many things, they simply were far from ideal for database work. Database programs took too long to write. The code required was unnecessarily lengthy. Each database vendor's product had a proprietary means of being accessed. The skill level required for database work was too high.

Structured Query Language was a brilliant solution to the database programming problem. Lengthy code was replaced with short SQL queries. Database programming became much faster and simpler. Differences between database products were hidden behind a layer of abstraction, making it possible for companies to switch database vendors or even intermix them without concern. With SQL, a more junior level of person could do database programming than in the past. Despite its name, SQL became more than just a *query* language: it eventually became possible to perform database *actions* as well. The success of SQL can't be denied. Over 25 years later, it continues to be pervasive. Today, we take it for granted.

We can draw an exact parallel between the database programming problems of yesteryear and the network/Internet programming problems of today. The symptoms are the same. Today's situation cries out for a similar solution to SQL. NQL was created to be that solution. While NQL is not grammatically similar to SQL, it shares SQL's characteristics: programs are short; code is easy to write and easy to read; development is rapid; and even very junior people can use it.

### III. Introducing NQL

NQL is a scripting language for developing connected applications. It has the right building blocks for bots and agents. There's a strong emphasis on communications, data conversion, automation, and intelligent behavior. Before going any further, let's see what some NQL code looks like. Listing 1 shows a sample script that retrieves news articles from the CNN main page and outputs headlines and links as XML. We'll trace through the script line-by-line later on, but even at first glance you should find it easy to follow.

```
get "http://www.cnn.com"
match '<li><a href="{link}">{headline}</a>'
while
{
    output headline, link
    nextmatch
}
```

*Listing 1—retrieves news headlines from a web site and outputs headlines and article links in XML*

In order to appeal to non-programmers as well as programmers, NQL stresses simplicity. Statement keywords are made up of words found in the dictionary. Odd punctuation is avoided. The syntax of the language is simple: one statement per line. If a statement requires multiple parameters, they are separated by commas. String constants are enclosed in single quotes or double quotes. Lines beginning with *//* are single-line comments. Multi-line */\** and *\*/* comments can also be used as in C, C++, and Java.

Like most languages, NQL supports standard control flow statements including *if*, *for*, and *while*. Curly braces are used to enclose statement blocks. One of the control flow statements is *thread*, which executes a block of code in a separate thread of execution.

NQL variables are not strongly typed and may contain any kind of data. You can require variables to be declared or not as you see fit, and you can choose from three kinds of error handling: program halt, executing error trapping code, or ignoring errors altogether. In addition to variables, NQL can also hold data on a stack. The stack is what allows NQL's statements to interact with each other smoothly.

NQL supports an impressive range of communications protocols, built-in and ready to use. The protocols available include HTTP and HTTPS for web access; FTP for moving files; MAPI, POP3, and SMTP for sending and reading e-mail; ODBC, JDBC, OLEDB, and ADO for database access; NNTP for reading newsgroups; LDAP for accessing directory servers; SNMP for network monitoring; and TELNET for interacting with legacy systems. Having oodles of protocols built-in to the language is only half the story. Communications in NQL is accomplished

with a small set of statement keywords designed to logically fit together. Differences in protocols are hidden. For example, the method of reading MAPI e-mail and the method of reading POP3 e-mail is the same. To get a feel for this, examine Listings 2a, 2b, and 2c. The code in Listing 2a reads unread e-mail. The code in Listing 2b reads through the articles in a newsgroup. The code in Listing 2c logs on to a Linux system and interacts with it.

```
openmail
firstunread
while
{
    ...do something with mail message...
    nextmessage
}
closemail
```

*Listing 2a—reading unread mail messages*

```
opennews "news.server.com", "alt.news.topic"
firstarticle
while
{
    ...do something with newsgroup article...
    nextarticle
}
closenews
```

*Listing 2b—reading articles from a newsgroup*

```
openterm "linuxserver2"
waitterm "login:"
sendterm "jsmith\r"
waitterm "password:"
sendterm "johnny\r"
waitterm "$ "
clearterm
sendterm "ls -l\r"
waittermquiet 2
getterm
closeterm
...do something with captured session data...
```

*Listing 2c—interacting with a legacy system and capturing a directory listing*

NQL scripts can work with many kinds of information, including text, web pages, XML, and multimedia types. NQL's variables and stack can hold string or binary information of arbitrary size. NQL has particularly strong support for XML, with direct capabilities for reading, parsing, and writing XML.

Since connected applications frequently need to be automated, NQL includes features for scheduling. Scripts can be set to execute at a specific date and time, and to run again at regular

intervals. Scripts code can wait for a specific time to arrive, wait for a specific interval to pass, or repeat execution of a code block at regular intervals. Scripts can declare time limits.

For intelligent behavior, NQL supports practical outpouring of AI research, including fuzzy logic and neural networks. The use of fuzzy logic can greatly improve the decision making of agents. NQL's fuzzy logic support includes language constructs for declaring condition evaluation functions and performing logical operations on fuzzy values. Neural networks have great application in predictive systems and pattern recognition. NQL makes it simple to create, train, save, and run neural networks, as Listing 3 illustrates.

```
////////////////////////////////////  
////////////////////////////////////  
//  
// Script name: geology  
//  
// Description: Demonstrates use of a neural network in the area  
of geology.  
//           Given indirect measurements, determines  
lithology.  
//           Inputs:  Gamma Ray, Neutron, and Density.  
//  
// Inputs:      Gamma Ray log value, Neutron log value, and  
Density log value  
//  
// Outputs:     Identification (Shale, Dolomite, Limestone,  
Sandstone)  
//  
// Notes:      The first time the script is run, the network  
will be created,  
//           trained, and saved (in geology.nnf). Thereafter,  
running the  
//           script will process input and generate output.  
//  
//           The file required for training is geology.trn  
(text data file).  
//           The file required for processing is geology.inp  
(text data file).  
//           Results are output to the file geology.out (text  
data file).  
//  
////////////////////////////////////  
////////////////////////////////////  
  
//if this is the first time this script is being run,  
//create, train, and save the neural network  
  
lookup "geology.nnf"  
else  
{  
    setneural 0.45, 0.90  
    createneural "backprop", 3, 4, 4
```

```

    trainneural "geology.trn"
    saveneural "geology.nnf"
    closeneural
    show "The network has been created and saved. Run the
script again to process data."
    end
}

//open previously saved neural network and run it against real
data

openneural "backprop", "geology.nnf"
processneural "geology.inp", "geology.out"
closeneural
show "The network has been run against the input data. Processing
is complete."

```

*Listing 3—creating, training, saving, and running a neural network*

#### IV. Interesting Aspects of NQL

Paradoxically, NQL's compactness of code is due in part to two notions that are borrowed from assembly language: stacks and condition codes. A stack is a list of values, kind of like a stack of dinner plates. A new value may be "pushed" onto the stack. The top value on the stack may be "popped" off of the stack. Unlike variables, which hold information by name, a stack can hold lots of information as a temporary holding area. For connected applications, stacks can be useful for recursive operations such as crawling web sites. NQL uses the stack as a way to let multiple statements work on the same piece of information.

NQL allows every statement to set a "success" or "failure" condition. For example, a pattern match statement results in a successful condition if a match is found, or a failed condition if a match is not found. Condition codes allow NQL's statements to be combined with control flow statements in a compact way.

Now that stacks and the success/failure condition code have been introduced, let's go back to Listing 1 and go through the code line by line. In line 1, a *get* statement retrieves a web page. Where does the web page go? On the stack. Line 2 is a *match* statement, which searches for its pattern in the top item on the stack, which just happens to be the HTML web page that was just retrieved. If the pattern match was successful, three things happen: the variable names in the pattern (*link* and *headline*) are set to values from the web page; the HTML on the stack is shortened in anticipation of more pattern matching; and a success condition is set. Line 3 is a *while* statement, which may look a bit odd since no condition is specified. This means that the success/failure condition (set by the previous statement) is what determines whether or not the block of code is executed. In the code block, line 5 contains an *output* statement which outputs the variables *headline* and *link* as XML. Line 6 is a *nextmatch* statement, which repeats the previous pattern match—this time, on the remainder HTML on the stack, which will find the next match. Like the *match* statement in Line 2, *nextmatch* will set variables and a success condition if it finds another match, and will again shorten the HTML on the stack. Thus, the *while* loop continues as long as there are matches. That's a lot of explanation, but the code itself is quite small and fairly clear, even to someone who hasn't previously been exposed to NQL.

The action of the *match* statement deserves some explanation. The pattern includes HTML tags and text, but may also contain wildcards. Although an asterisk may be used to indicate a wildcard area, it is more common to specify a variable name enclosed in curly braces. When a match occurs, variables are created automatically with these names that contain data from the pattern match. This is called *automatic variable mapping*. Database queries also utilize automatic variable mapping, as Listing 4 shows. For each record retrieved, the fields are moved into variables and are ready for immediate use.

```
opendb "customers"
select "SELECT Customers WHERE BALANCE > 0.00;"
while
{
    show CustID, Company, Addr1, Addr2, City, State, Zip, Phone
    nextrecord
}
```

NQL is designed to play a role wherever it is needed. An NQL script can run on the desktop, on a middleware system, on a server, or on a web server as a CGI application. You can also elect to combine NQL with another programming language. Virtually all languages, including VB, ASP, C++, and Java, can call NQL as a component. This permits you to continue using your preferred development tools and still tap into the power of NQL when you need it.

NQL can be used to drive web applications. NQL program code and HTML can be interspersed, just as is the case with ASP and JSP. In this context, NQL provides mobile device support for Palm VIs, Internet-enabled smart phones, and Pocket PCs. NQL has built-in features for recognizing these devices and responding appropriately in their native content language. This means that a single NQL script can be accessed through a myriad of devices. NQL can also detect these devices' serial numbers, which is handy for mapping them to specific user IDs.

## V. Summary

Network Query Language was designed for same-day agent development, and it delivers on that promise. The built-in functionality, simple grammar, and compact design create a measurably faster environment for realizing connected applications. The best way to appreciate this is to experience it in person: a 30-day trial of NQL is available at <http://www.nqli.com>.

NQL is currently available for Windows, Windows NT, and Windows 2000. There is also a Java edition of NQL nearing release, which is initially being qualified for Linux, the Macintosh, and Solaris. NQL Inc. is also contemplating porting NQL to popular PDA platforms, including Palm and Windows CE.