

SQL*NET MESSAGE FROM CLIENT...IDLE TIME OR YOUR PROBLEM?

Gary Goodman, Hotsos Enterprises, Ltd.

INTRODUCTION

An interesting result from our research at Hotsos is that we continually find ourselves proving that some old – and some recent – standards and rules of Oracle optimization are invalid. The shift from system-wide data and ratios to a user response time focus utilizing *Oracle timed events* (the 10g name for what we used to call the *wait interface*) has provided the new clarity. The goal of this paper is to properly define the *SQL*Net Message from client* event, show when it is a problem, and to discuss the root cause and remedy for a few of the related performance issues we have seen in the field.

DEFINING THE SQL*NET MESSAGE FROM CLIENT EVENT

Before you can understand if *SQL*Net Message from Client* is your problem, you must first understand what the event represents. The short answer is that it is the time spent between database calls (for example, *parse*, *execute*, and *fetch*). If you are looking at extended SQL trace data, you will often see the following pattern:

<i>SQL*Net message from client</i>	(available and waiting to do some work)
<i>Parse</i>	(working on the <i>parse</i> call)
<i>SQL*NET message to client</i>	(acknowledgement that the <i>parse</i> call is complete)
<i>SQL*Net message from client</i>	(available and waiting to do some work)
<i>Execute</i>	(working on the <i>exec</i> call)
<i>SQL*NET message to client</i>	(acknowledgement that the <i>exec</i> call is complete)
<i>SQL*Net message from client</i>	(available and waiting to do some work)
<i>Fetch</i>	(working on the <i>fetch</i> call)
<i>SQL*NET message to client</i>	(acknowledgement that the <i>fetch</i> call is complete)
<i>SQL*Net message from client</i>	(available and waiting to do some work)

Given this pattern, you can see that the counts for *SQL*Net message from client* and database calls will be nearly the same. I will explain why this is important knowledge later.

To the operating system, *SQL*Net message from client* is simply a *read* call. It reflects the Oracle kernel waiting for a properly formatted database call to work on.

To better understand this definition, a system call tracer (*strace* for a Linux server) shows the OS view of your idle process. What you will see is a *read* call waiting for input.

```
B 17:27:19 $ strace -r -p1368
0.000000 read(7,
```

Note: The '7' is the file descriptor on which the read will be acting.

When a *parse*, *execute*, or *fetch* call appears, a *gettimeofday* OS call timestamps the end of the *SQL*Net message from client* event, and Oracle starts working. When the database call is complete, a *SQL*Net message to client* acknowledges that the work is complete—the *SQL*Net message to client* event, as you might have expected, is the event that is associated with an OS *write* call. Then a new *SQL*Net message from client* event begins.

The following *strace* output gives you a good picture of the complete cycle.

```
B 17:27:19 $ strace -r -p1368
0.000000 read(7, "\0\345\0\0\6\0\0\0\0\0\21i\32\304\315\22\10\1\0\
0\0\1\0"... , 2064) = 229
20.465314 gettimeofday({1065738560, 69183}, {300, 0}) = 0
0.000201 gettimeofday({1065738560, 69377}, {300, 0}) = 0
0.000181 gettimeofday({1065738560, 69558}, {300, 0}) = 0
0.000205 getrusage(RUSAGE_SELF, {ru_utime={0, 60000}, ru_stime={0, 10000},
...}) = 0
...
0.000167 stat64("/u01/oradata/admin/V901/udump/ora_1368.trc", 0xbfffae8c) = -1
ENOENT (No such file or directory)
0.000258 open("/u01/oradata/admin/V901/udump/ora_1368.trc",
O_WRONLY|O_CREAT|O_TRUNC, 0660) = 5
0.000247 write(5, "", 0) = 0
0.000172 write(5, "/u01/oradata/admin/V901/udump/or"... , 42) = 42
...
0.000179 write(5, "WAIT #1: nam='\SQL*Net message to"... , 73) = 73
...
0.000043 gettimeofday({1065738579, 946088}, {300, 0}) = 0
0.000060 write(10,
"\0\202\0\0\6\0\0\0\0\4\1\0\0\0\1\1\0\0\0\0\5\0\0\0\0\1"... , 130) = 130
0.000753 read(7,
```

ORIGIN OF 'IDLE TIME' LABEL

It is easy to see where the idle time label has originated. *SQL*Net message from client* (as well as some other timed events) happen between database calls [Millsap (2003)]. It must be idle if the database is not doing anything!

The answer resides in the history of Oracle optimization methods. For years we were taught to collect and analyze system-wide data. We were to keep ratios above certain guidelines, totals within certain boundaries, and averages above or below accepted baselines.

With the addition of *timed events* in Oracle 7.0.12, performance analysts had a new tool to capture data for more than just database calls. Suddenly we could see everything (well, some in version 7, most in version 8, virtually all in version 9+) that the database was doing or waiting on! For *timed events* occurring during database calls, we could all agree that this represented relevant Oracle kernel activity. System-wide evaluation of *timed events* between database calls became much more of a problem. How much of the system-wide total was truly idle time and how much represents user response time? Is it 90%-10%? Is it 10%-90%? It is impossible to tell. This dilemma leads to misguided advice like I found with a simple Google search using the term 'idle event'.

EXAMPLE I [BURLESON –WAIT]

"We also have to filter out wait events that are not helpful to our tuning effort. The list below shows all system "idle" wait events that have no meaningful information:

- *dispatcher timer*
- *lock element cleanup*
- *Null event*
- *parallel query dequeue wait*
- *parallel query idle wait - Slaves*
- *pipe get*
- *PL/SQL lock timer*
- *pmon timer*

- *rdbs ipc message*
- *slave wait*
- *smon timer*
- *SQL*Net break/reset to client*
- *SQL*Net message from client*
- *SQL*Net message to client*
- *SQL*Net more data to client*
- *virtual circuit status*

EXAMPLE #2 [BURLESON –BOTTLENECK]

“The next prerequisite to using bottleneck analysis is that certain wait events should be filtered out of any metrics used to diagnose performance bottlenecks. For example, Oracle will record a wait statistic that represents how long a particular user sits at their SQL*Plus prompt between each issued database request.

Such a statistic provides no real value to a DBA who is trying to figure out where a database bottleneck exists. Any SQL scripts that are used to collect database wait statistics should exclude such events. A listing of these Oracle events (normally dubbed “idle” events) to eliminate includes:

...

- *SQL*Net message from client*
- *SQL*Net break/reset to client*
- *SQL*Net message to client*
- *SQL*Net more data to client*

...”

My point here is not to disparage other authors, but to simply highlight the fact that these conclusions were drawn assuming performance analysts only collect system-wide data. We increasingly see *SQL*Net Message from Client* as the top response time contributor and it is a significant mistake to discard this data as ‘idle time’.

COMMON ROOT CAUSE ISSUES

The contribution of any event to your response time is a function of the number of times it happens (event count) and the average duration per event (average duration).

Event Count x Average Duration = Event’s Contribution to Response Time

Given this, response time improvement comes down to two simple tasks. The first is to lower the count for your top response time contributors. One of my favorite Anjo Kolk quotes is “The fastest way to do something is to not do it”. If your code currently requires 10,000 *SQL*Net message* events, then requiring anything less than 10,000 will be quicker.

Once you have reduced your event count, your second task is to ensure that your duration per event is optimized. While this seldom has the same impact as count reduction, you will often find positive ROI for tasks aimed at reducing average duration.

The remainder of this paper is dedicated to diagnosing and repairing performance issues where *SQL*Net message from client* is the largest contributor to response time.

HIGH EVENT COUNT #1

THE PROBLEM: *SQL WITH LITERAL VALUES NOT REUSEABLE*

One of the most misunderstood performance issues we see is the unnecessary workload created when application code does not take advantage of bind variables. Consider the following SQL statement. This is a batch job example from a customer’s production system. The job was a separate executable per the “client-server” model and used no stored procedures.

The top contributor to the response time: *SQL*Net message from client*

```

INSERT INTO STAGING_AREA (
    DOC_OBJ_ID,
    TRADE_NAME_ID,
    LANGUAGE_CODE,
    OBJECT_RESULT,
    GRAPHIC_FLAG,
    USER_LAST_UPDT,
    TMSP_LAST_UPDT)
VALUES (
    1000346,
    54213,
    'ENGLISH',
    '<BLANK>',
    'N',
    'sa',
    TO_DATE('11/05/2001 16:40:54', 'MM/DD/YYYY HH24:MI:SS')
)

```

This is a simple insert statement that has virtually no chance of ever being reused by Oracle. The call detail data below highlights this point.

Cursor Action	Library Misses	Action Count	Rows Processed
Parse	348	348	0
Execute	0	348	348
Fetch	0	0	0
Total	348	696	348

This statement was parsed 348 times with 348 library cache misses. It could never be reused because it could never find an exact match – including all 7 literal values – in the library cache. 348 *parse* calls means 348 *SQL*Net message from client* events.

THE REMEDY: USE BIND VARIABLES!

Consider this revised version of the above SQL statement:

```

INSERT INTO STAGING_AREA (
    DOC_OBJ_ID,
    TRADE_NAME_ID,
    LANGUAGE_CODE,
    OBJECT_RESULT,
    GRAPHIC_FLAG,
    USER_LAST_UPDT,
    TMSP_LAST_UPDT)
VALUES (
    :A1,
    :B1,
    :C1,
    :D1,
    :E1,
    :F1,
    TO_DATE(:G1, 'MM/DD/YYYY HH24:MI:SS')
)

```

By removing the literal values in this insert, the code can now parse this statement once and then *execute* it many times inside of a loop with different bind variable values. The resulting call detail looks like this:

Cursor Action	Library Misses	Action Count	Rows Processed
Parse	1	1	0
Execute	0	348	348
Fetch	0	0	0
Total	1	349	348

The addition of bind variables reduces the *parse* call activity down from 348 to 1; a reduction of 347 database calls AND a reduction of 347 *SQL*Net Message from client* events.

HIGH EVENT COUNT #2

THE PROBLEM – NO ARRAY PROCESSING

Let's look back at the last example (#1 above). We have already reduced the call volume significantly by using bind variables, but we still have 348 executes processing 348 rows.

Cursor Action	Library Misses	Action Count	Rows Processed
Parse	1	1	0
Execute	0	348	348
Fetch	0	0	0
Total	1	349	348

Since we know this is a batch job with no user interaction, do we really need one execute per row?

THE REMEDY:

Use array processing (*execute* and *fetch* calls) where possible. With array executes of 100, we can reduce the 348 executes to 4 calls with 100, 100, 100, and 48 rows processed.

Cursor Action	Library Misses	Action Count	Rows Processed
Parse	1	1	0
Execute	0	4	348
Fetch	0	0	0
Total	1	5	348

The addition of array executions reduces the *execute* call count down from 348 to 4; a reduction of 344 more calls AND a reduction of 348 more *SQL*Net Message from client* events.

HIGH EVENT COUNT #3

THE PROBLEM: COORS TWINS GOOD – PARSE TWINS BAD

Hopefully by now you agree that unnecessary parsing damages your performance. The following is a second example of unnecessary parsing that had us all confused when we first encountered it. Below are the call details for one of many SQL statements from this client's trace file.

The top contributor to the response time: *SQL*Net Message from client*

Cursor Action	Library Misses	Action Count	Rows Processed
Parse	303	606	0
Execute	0	303	303
Fetch	0	0	0

Total	303	909	303
-------	-----	-----	-----

It is bad enough to parse once per execution, but this code parses TWICE for every execution. Why would anyone require 606 *parse* calls for only 303 executions? This increases the total database calls by 2 x *parse* count AND increases the *SQL*Net Message from client* count by the same amount.

THE REMEDY: CHANGE HOW YOU COMPILE!

The answer here lies within a feature designed to make a developer's job easier. If a *parse* call fails during unit test, a bland 'parse failed' message does little to help you determine why it failed. Most development tools can be configured to show you exactly why a *parse* call error occurred (i.e. the exact column or table name you misspelled). They accomplish this by adding a *describe* before the *parse* call. As you might have guessed, this *describe* requires a *parse* call of its own leaving you with twin *parse* calls for every execution.

The remedy is to turn this feature off before you compile your code. How you do this is different depending on the tool you use. We have come across this problem with both PowerBuilder and Perl.

HIGH AVERAGE DURATION #1

THE PROBLEM – MISSING TNSNAMES.ORA ALIAS

Occasionally we find customers who simply are not taking advantage of all of the performance features available to them. I will demonstrate this using the data from a slow payroll batch job. As you can see from the trace data analysis below, the bottleneck was clearly *SQL*Net message from client* and *SQL*Net more data from client*. Collectively they accounted for 70.7% of the total run time.

Response Time Component	Duration		# Calls	---- Duration Per Call ----		
				Avg	Min	Max
SQL*Net message from client	984.01s	49.6%	95,161	0.010340s	0.00s	0.31s
SQL*Net more data from client	418.82s	21.1%	3,345	0.125208s	0.00s	0.27s
...						
Total	1,985.40s	100.0%				

From discussions with the customer – and from data in the trace file – we knew that the payroll code and the database were on the same server. Given that, 10 milliseconds seemed like a very high average duration for network roundtrips. It turns out the account had a standard *tnsnames.ora* file configured with one alias for *PROTOCOL=TPC* that was used on all servers. The TPC protocol must use the network interface card (NIC) on the server to route its traffic to the appropriate IP address. What this means is that for each of the 95,161 *SQL*Net message from client* and 3,345 *SQL*Net more data from client* events, the packets went to the NIC, up through as many layers as necessary to resolve the IP address, determined "Oh, that's me" and were routed back down to the same server they came from.

THE REMEDY:

There are two protocols available to remove the NIC from the above problem. When the code and the Oracle kernel are on the same server, you can add an additional alias that will skip the NIC and handle the transaction directly in memory. The two options for this alias configuration are *PROTOCOL = IPC* or *PROTOCOL=BEQ*. The differences between the two are well documented at a variety of locations on the internet.

HIGH AVERAGE DURATION #2

THE PROBLEM – COMPETITION CREATES BOTTLENECK

Occasionally you will find a high average duration but there is nothing in your Oracle configuration or your program code that is bad. The problem is a capacity issue on your network. It might be system-wide or it might only be tied to a specific router.

THE REMEDY:

This is one of those unique opportunities where you might get to point your finger at the network administrator and say ‘You have a problem with the network’. In reality, it may be one of two problems – or more likely, a bit of both.

There may be other users on the network who have ‘chatty’ applications. Remember our high event count examples from above? Problems like these drive unnecessary network traffic and consume capacity. Enough of these and network bottlenecks emerge and queue time is added to service time. Enough queuing time and your average round trip durations go up.

The problem may also be that your network, or some portion of it, is under capacity for the efficiently coded workload it has to handle. This will cause the same delays discussed above. As mentioned earlier, the problem is probably some of both.

Network Administrator Disclaimer: I’m not trying to make your life miserable – honest. Hopefully this paper will give you more ammunition when people with unnecessarily ‘chatty’ applications try to blame you!

CLOSING

By now, I hope you are convinced that a large *SQL*Net message from client* event count is a sign of a potentially significant problem and not something to be taken lightly. Any sizeable component of a user’s response time requires investigation. The key is not to get caught up in system-wide statistics that will often times hide the root cause of your current performance issue.

REFERENCES

- C. Millsap. 2003. Oracle Operational Timing Data. Hotsos. Available at www.hotsos.com.
- D. Burleson. Oracle Wait Analysis Techniques. <http://www.dbazine.com/burleson8.html>
- D. Burleson. Oracle Bottleneck Analysis. http://www.remote-dba.cc/teas_aegis_9.htm