Connection Management in an Oracle RAC Configuration

James Morle
(James.Morle@scaleabilities.co.uk)

A look at the various types of Load Balancing and Connection Failover in a RAC environment

INTRODUCTION

A lot of confusion exists about the management of connections in a Real Application Clusters configuration. Some even take the viewpoint that RAC is like a fault-tolerant appliance with a single, always-on IP address. While this certainly is not the case, even those who get closer to the truth come out confused about why their load balancing isn't working and why their connections just don't seem to failover properly. The author of this paper found himself starting to get confused too, given the increasing functionality in the area of RAC session management, so it was time to do a little investigation and come up with some real facts.

One thing that is very apparent from the outset is that the documentation for this area is somewhat scattered across the Oracle documentation library. It might be all there, it might not: I can't tell because there is no point-source of the truth. In this age of the search-engine, I simply don't have the patience anymore to read every manual from cover to cover! So, with apologies to the Oracle documentation team, I present here the output from a subtle blend of documentary fact, natural curiosity and actual experimentation.

THE TEST SYSTEM

A few years ago I was doing so much RAC experimentation that I figured it was time to build a fairly decent RAC Cluster of my own in my office. More accurately, it was time to build one in my *garage*, because there was no way on earth that I was going to let **that much** noise pollution into my carefully silenced domain. So, I installed a 42U rack into the garage, installed a pair of fibre-optic cables between my office and the garage, and started to install the hardware.

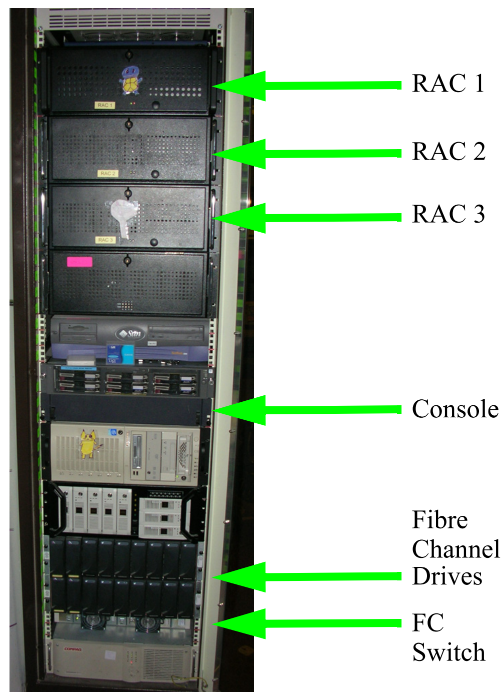
It seemed appropriate that I should ressurect the cluster for this testing. I played with a bunch of weird configurations before that decision, such as VMWare Team clusters and iSCSI-based clusters, but decided to build a cluster that was as close as possible to the kind of configuration that my customers use.

I built the nodes using various old PC hardware, with quite a variation in power and ability. The node configurations look like this:

TABLE 1. Node Configurations

Node Name	CPU Count	CPU Type	Memory
RAC1	2	450 MHz Pentium II	1024MB
RAC2	2	1.2 GHz Pentium III	1024MB
RAC3	1	850 MHz Pentium III	640MB

And the actual rack looks like this:



Note, only the front of the rack is fit for human consumption. The rear of the rack would make any networking guy freak out, so it's best to keep that view a secret.

Each of the servers is fitted with a pair of Intel Pro/1000 Gigabit Ethernet cards, which gives the machines the possibility to boot over the network using PXE. All the

machines were installed this way, using Kickstart and Centos 4.3 (based upon Red-Hat EL4U3). I elected to invest the time in making the build image as complete as possible, because I have a habit of totally destroying nodes in a RAC cluster once I start doing the more punishing testing. Hopefully, though, I can be a little kinder on this occasion so a complete rebuild won't be necessary.

Each server was fitted with a Qlogic Fibre Channel Adapter, and connected into the SAN. The SAN is composed of an EMC DS-16b Fibre Channel Switch and some old Fibre Channel drive shelves from an old NetApp Filer. This doesn't make it a particularly feature rich SAN - it simply makes the raw drives visible to each node in the cluster, without any fancy stuff like RAID.

The machines are wired with two separate networks:

- 192.168.1.0/24 - the internal network at Scale Abilities
- 10.10.1.0/24 - the private, unrouted, cluster interconnect

Once the cluster was physically built, installed and tested from a generic O/S stand-point, I proceeded to install Oracle. I elected to go for the 100% Oracle stack, using Oracle Clusterware, Automatic Storage Manager (ASM) and of course the RDBMS. All products were 10g, v10.2.0.2.

I then built a generic database on top using DBCA. That was quite interesting, seeing that run on the 2x450MHz RAC1 node - the machine was completely cpu-bound for quite some time, load average of 14, just running the Java application! Oh well, I guess nobody runs TPC benchmarks on installer products.

The database was named, imaginatively, RACDB. The instances were called INST1, INST2, and INST3. I wouldn't normally recommend having totally different names for the instances than the database (i.e. RACDB1, RACDB2, RACDB3), but I wanted to have an immediate distinction between the database and the instances for the sake of this paper.

So, that's it, everything is built, and ready for running some tests.

Load-Balancing in RAC Environments

TEST ONE - SERVER-SIDE LOAD BALANCING

Practically every client site I go into has a **tnsnames.ora** file like this:

```
RACDB =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = rac1-vip.sa.local)(PORT = 1521))
    (ADDRESS = (PROTOCOL = TCP)(HOST = rac2-vip.sa.local)(PORT = 1521))
    (ADDRESS = (PROTOCOL = TCP)(HOST = rac3-vip.sa.local)(PORT = 1521))
    (LOAD_BALANCE = yes)
    (CONNECT_DATA =
      (SERVER = DEDICATED)
      (SERVICE_NAME = RACDB)
    )
  )
```

It's not surprising, really, because this is exactly what the installer (or netca) will create for you. However, I think this is where a lot of the initial confusion comes from, particularly in terms of the **LOAD_BALANCE=YES** directive in the TNS entry. This is actually *client-side load balancing*, which we will look at in a moment. For the time being, though, we are going to simplify things and adopt a much more simplistic configuration:

```
RACDB =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = rac2-vip.sa.local)(PORT = 1521))
    (LOAD_BALANCE = no)
    (CONNECT_DATA =
      (SERVER = DEDICATED)
      (SERVICE_NAME = RACDB)
    )
  )
```

So, we've turned off client-side load balancing, and picked one of the nodes to go after that initial connection. On my test client machine, I now run the following script:

```
a=100
while [ $a -gt 0 ]
do
       sqlplus myuser/myuser@RACDB &
       a=$((a-1))
       sleep 1
done
```

Not a terribly clever script - it just logs into **sqlplus** a hundred times. Now let's take a look what happened using **GV$SESSION**:

```
  1  select inst_id,count(*)
  2    from gv$session
  3   where username='MYUSER'
  4*  group by inst_id
SQL> /

   INST_ID   COUNT(*)
---------- ----------
         1         33
         2         34
         3         33
```

Wow! Perfect load balancing across the three nodes! Let's take a look at what the listener thinks is going on, first from RAC2:

```
Service "RACDB" has 3 instance(s).
  Instance "INST1", status READY, has 1 handler(s) for this service...
    Handler(s):
```

```
      "DEDICATED" established:33 refused:0 state:ready
         REMOTE SERVER
         (ADDRESS=(PROTOCOL=TCP)(HOST=rac1.sa.local)(PORT=1521))
  Instance "INST2", status READY, has 2 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:34 refused:0 state:ready
         LOCAL SERVER
      "DEDICATED" established:0 refused:0 state:ready
         REMOTE SERVER
         (ADDRESS=(PROTOCOL=TCP)(HOST=rac2.sa.local)(PORT=1521))
  Instance "INST3", status READY, has 1 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:33 refused:0 state:ready
         REMOTE SERVER
         (ADDRESS=(PROTOCOL=TCP)(HOST=rac3.sa.local)(PORT=1521))
```

Then from RAC1:

```
Service "RACDB" has 3 instance(s).
  Instance "INST1", status READY, has 2 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:0 refused:0 state:ready
         REMOTE SERVER
         (ADDRESS=(PROTOCOL=TCP)(HOST=rac1.sa.local)(PORT=1521))
      "DEDICATED" established:33 refused:0 state:ready
         LOCAL SERVER
  Instance "INST2", status READY, has 1 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:0 refused:0 state:ready
         REMOTE SERVER
         (ADDRESS=(PROTOCOL=TCP)(HOST=rac2.sa.local)(PORT=1521))
  Instance "INST3", status READY, has 1 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:0 refused:0 state:ready
         REMOTE SERVER
         (ADDRESS=(PROTOCOL=TCP)(HOST=rac3.sa.local)(PORT=1521))
```

So, to paraphrase that output: RAC1 can 'see' 33 direct connections made to it's local listener, and can see nothing else. RAC2, on the other hand, can see 34 direct connections but also 33 each to RAC1 and RAC3 as "REMOTE SERVER" connections. These are referrals by the RAC2 listener to the other nodes in the cluster, essentially punting the connection elsewhere. So, is RAC2 still involved in the connections that were punted off to the other nodes? Let's look at **netstat** on the client machine:

```
morlej@pikachu:~> netstat -a | awk '/:ncube-lm/ { print $5 }' | sort | uniq -c
     33 rac1.sa.local:ncube-lm
     34 rac2-vip.sa.lo:ncube-lm
     33 rac3.sa.local:ncube-lm
```

Normally I would use the '**-n**' option to **netstat** and **grep** for the SQL*Net port (1521) instead of the obscure '**ncube-lm**' service, but I wanted to get the hostnames resolved for clarity in this paper. It's kind of funny that /etc/services in Linux has port 1521 mapped to be **nCube License Manager** instead of SQL*Net - it's been a while since I needed a license for a 90's proprietary MPP machine on my Linux server. Anyway, moving on...

So, the connections are made directly between the client and the instance that the session resides upon. BUT, did you notice the problem in that output? The redirected sessions that were sent to RAC1 and RAC3 were redirected to the physical IP address of the machine, not the Virtual IP (VIP). This is Very Bad News™, because if one of those nodes were to fail then the session would have towait for the full TCP timeout period before the connection was dropped. This is typically somewhere between five minutes and an hour - an eternity in either case - and certainly not a

desirable situation when Oracle has gone to the trouble of making faster methods of failover that require the use of the VIP!

This turns out to be a little bit of a bug in RAC, but luckily there is a workaround for this. Remember that this whole configuration was built using 100% assistants? Well, it looks like they miss a mandatory parameter setting in the database instances to give the listeners a clue about which IP address they should be punting off to[1].

When the **PMON** process registers with the listener, it defaults to using the main IP address of the host (or more specifically, the IP address for the hostname of the machine). In a RAC environment, and also in HA Clusters that use virtual IP addresses, the physical IP address of the machine is not the correct one to use, and so PMON needs a clue.

Let's take a quick look at the listener.ora file on RAC2:

```
LISTENER_RAC2 =
  (DESCRIPTION_LIST =
    (DESCRIPTION =
      (ADDRESS = (PROTOCOL = TCP)(HOST = rac2-vip.sa.local)(PORT = 1521)(IP =
FIRST))
      (ADDRESS = (PROTOCOL = TCP)(HOST = 192.168.1.17)(PORT = 1521)(IP = FIRST))
    )
  )

SID_LIST_LISTENER_RAC2 =
  (SID_LIST =
    (SID_DESC =
      (SID_NAME = PLSExtProc)
      (ORACLE_HOME = /opt/oracle/db10.2)
      (PROGRAM = extproc)
    )
  )
```

The reason that **PMON** is able to register with the local listener is highlighted in the file: It is listening on the physical IP address in addition to the VIP. We could remove this reference, which would prevent the listener from listening on that address, and thus prevent **PMON** from registering on it. Unfortunately, this would have two negative side-effects:

- PMON would fail to register *at all* with the local listener
- The **lsnrctl** commands (such as **lsnrctl services**) would fail because they default to the implicit **listener.ora** entry for a listener called **LISTENER**, which sends commands on the physical IP address, port 1521. If there is no listener listening on the physical IP address, we would need to always specify the name of our VIP listener (e.g. **lsnrctl services LISTENER_RAC2**), which is just too much typing for me!

The best fix, which addresses both these problems, is to set the instance-specific database parameter **LOCAL_LISTENER** in the **init.ora** (or spfile) to connect to the local listener using it's VIP instead of the physical IP[2]:

```
alter system set local_listener=
       '(ADDRESS = (PROTOCOL = TCP)(HOST = rac1-vip.sa.local) (PORT = 1521))'
       scope=both sid='INST1';
```

1. Please reference Metalink Note 364855.1 for further background
2. This parameter can also refer to an entry in the **tnsnames.ora** insetad of the full TNS specification

Doing it this way, we get to keep the convenience of just addressing the listener by default commands, and **PMON** gets to register. Note, though, that if you have multiple VIPs and instances on the same host, it is necessary to remove the reference to the physical IP in the listener configuration and address them by name.

After making this change, lsnrctl services looks like this:

```
Service "RACDB" has 3 instance(s).
  Instance "INST1", status READY, has 1 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:0 refused:0 state:ready
         REMOTE SERVER
         (ADDRESS=(PROTOCOL=TCP)(HOST=rac1-vip.sa.local)(PORT=1521))
  Instance "INST2", status READY, has 1 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:0 refused:0 state:ready
         REMOTE SERVER
         (ADDRESS=(PROTOCOL=TCP)(HOST=rac2-vip.sa.local)(PORT=1521))
  Instance "INST3", status READY, has 2 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:0 refused:0 state:ready
         REMOTE SERVER
         (ADDRESS=(PROTOCOL=TCP)(HOST=rac3-vip.sa.local)(PORT=1521))
      "DEDICATED" established:0 refused:0 state:ready
         LOCAL SERVER
```

At least the listener can now see all the correct hostname/IP addresses for the virtual IPs. Upon re-executing the test, I hit a new problem - all the connections were going to INST1 and INST3, none to INST2. I determined this to be caused by the listener trying to load balance based upon system load, and set the following parameter in the listener.ora on INST2:

```
prefer_least_loaded_node_LISTENER_RAC2=off
```

This parameter disables the CPU runqueue-based load distribution, forcing simple round-robin balancing. For sensibly designed applications that only connect at startup time (rather than frequently reconnecting), turning this parameter off is the right thing to do. The very act of connecting to the database causes load on the server, so it could even be the wrong thing to do for badly designed applications that frequently reconnect to the database. For sure, if you are using a multi-node connection pool and not making any attempt to partition the application workload across the RAC cluster by workload, this is the option to use.

Upon re-executing the test I had the following connection profile from the client's perspective:

```
     33 rac1-vip.sa.lo:ncube-lm
     34 rac2-vip.sa.lo:ncube-lm
     33 rac3-vip.sa.lo:ncube-lm
```

Great - we now have a simple and reliable server-side round-robin load balanced connection method. There's just one problem with it - if RAC2's listener goes away, we have no way to connect to the database! The answer, of course is to allow the client to try connections to different nodes...

TEST TWO - CLIENT-SIDE LOAD BALANCING PLUS SERVER-SIDE LOAD BALANCING

Now that the server-side is stable, let's re-introduce the references to the other nodes within the client-side **tnsnames.ora**:

```
RACDB =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = rac1-vip.sa.local)(PORT = 1521))
    (ADDRESS = (PROTOCOL = TCP)(HOST = rac2-vip.sa.local)(PORT = 1521))
    (ADDRESS = (PROTOCOL = TCP)(HOST = rac3-vip.sa.local)(PORT = 1521))
    (LOAD_BALANCE = yes)
    (CONNECT_DATA =
      (SERVER = DEDICATED)
      (SERVICE_NAME = RACDB)
    )
  )
```

Re-executing the connection test shows that the connections are distributed in just the same way as the last pure server-side load balancing test. Now let's stop the listener are rerun the test:

```
     50 rac1-vip.sa.lo:ncube-lm
     50 rac3-vip.sa.lo:ncube-lm
```

Fair enough - the listener is down on RAC2, so we can't connect there. This is a good time to point out that RAC load-balancing is a **connect-time only** feature. Re-enabling the listener on RAC2 will not result in any of these connections migrating to that node.

OK, that wasn't very exciting was it? That's because the client-side load balancing is being manipulated by the server-side load balancing to produce the same result as just server-side. All we gained here was the ability to connect to the cluster in the event that one of the nodes is unavailable.

TEST THREE - CLIENT-SIDE LOAD BALANCING ONLY

Let's make things more interesting, and disable server-side load balancing. I did this by simply setting the **REMOTE_LISTENER** parameter to an empty string on all instances and registering **PMON** with the local listener:

```
SQL> alter system set remote_listener='' scope=both;

System altered.

SQL> alter system register;

System altered.

SQL>
```

Let's check what the listener can see, too:

```
Service "RACDB" has 1 instance(s).
  Instance "INST2", status READY, has 1 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:0 refused:0 state:ready
         LOCAL SERVER
```

Splendid, it is only aware of itself. Now when we connect the same 100 sessions, what happens?

```
     30 rac1-vip.sa.lo:ncube-lm
     37 rac2-vip.sa.lo:ncube-lm
     33 rac3-vip.sa.lo:ncube-lm
```

This is the effect of client-side load balancing, which is described in the documentation as "try each address randomly". It seems like the randomness is somewhat uniform in distribution, but not quite a round-robin balance. Now let's change the **LOAD_BALANCE=yes** declaration in the client-side **tnsnames.ora** to **LOAD_BALANCE=no** and rerun the test.

The connection profile, perhaps unsurprisingly, looks like this:

```
100 rac1-vip.sa.lo:ncube-lm
```

The reason for this is that with **LOAD_BALANCE=no**, Oracle only ever tries the addresses in the order they appear in the **tnsnames.ora** file. In fact, if **FAILOVER** is set to “**off**” in the **tnsnames.ora** (defaults to “**on**”), Oracle will **only** try the first address, and if it fails will return an error.

LOAD-BALANCING CONCLUSION

The simple fact is, server-side load balancing is the superior option for RAC systems. The TNS configuration should be setup and tested with a single address in the client-side **tnsnames.ora**, then once the load balancing is working adequately, add the other addresses and turn on **LOAD_BALANCE** on the client. The effect of turning on client-side load balancing is minimal - it increases the chance that the connection will go to the appropriate node first, without needing a redirect from the server. That’s all it will do, in fact - the server will remain the master of the connection distribution. So, in summary, for well-designed applications that connect using pools, or at least infrequently:

- Set **PREFER_LEAST_LOADED_NODE_<listener_name>=off** in the **listener.ora** file on each RAC node
- Set **LOCAL_LISTENER** on each instance to point at the VIP on the local node
- Add all nodes to the client-side **tnsnames.ora**
- Set **LOAD_BALANCE=on** in the client-side **tnsnames.ora**

Now that we’ve got a stable and predictable load-balancing configuration, let’s move on to connection failover in a RAC environment.

Connection Failover in RAC Environments

First of all, let's define what connection failover is trying to achieve. Connection failover occurs when the specific database instance or node that the session is connected to experiences a failure. The actual failover is the act of reconnecting the failed client connection to a surviving instance, preferably with only a small delay to the end user, and no data loss. The failover procedure has three high level steps:

1. Failure Detection
2. Physical Failover
3. State Recovery

FAILURE DETECTION IN RAC CONNECTIONS

Failure detection is a simple concept, and yet one that is notoriously difficult to implement satisfactorily. The problem lies in the two conflicting requirements for effective failure detection:

1. Detection must be a fast as possible
2. Detection must not result in false positives

To highlight how these requirements conflict, let's look at an example. If we have a two node RAC cluster, and one of the nodes becomes very busy because of some rogue queries, what should the correct course of action be? The node is not 'down', and yet it is not servicing requests because it is totally saturated. According to the first requirement, we want to know if the node has failed as soon as possible. When does it become appropriate to make that determination? Five seconds? Fifteen seconds? Thirty? A minute?

Let's assume a reasonably fast detection of fifteen seconds for this example. So, after fifteen seconds of unresponsiveness, we start a failover procedure that could take thirty or more seconds. In addition, the unresponsive node will need to be forcibly shutdown to prevent it from potentially resuming work without permission, therefore necessitating a further step to reintroduce this node as a member of the cluster. That's all well and good, but what if the rogue queries that caused this condition finish in thirty seconds? All the failing over and failing back could have been avoided and the end-users would still have been back in service sooner than if we deemed the node 'down'. If I had just waited...

This is a classic "false positive" - a positive failure detection that was wrongly diagnosed and thus violated the second of the two requirements. Let's look at another example, using the same two-node RAC cluster. In this case, one of the nodes has experienced a kernel panic - it is never coming back from this one without a reboot. The symptoms of this failure are effectively identical to the last one - the node has become unresponsive to requests, including health monitors. This time we have configured the failure detection to wait for a full minute, to make sure that we don't experience the false positive scenario above. So we give the node a chance to recover for a full minute, a full minute in which the node will not recover. In this case, though we did not experience a false positive, we violated the first requirement, that of fast failure detection.

Failure detection is often something that has to be application and implementation specific; each environment is different and so the failure detection times should also be different, governed by the availability needs of the system.

For the sake of simplicity, let's just focus on the detection of failed network connections for the remainder of this section.

First of all, it is vital the the connections are all made via the VIP. If the connections are made to the physical IP, the detection of the failure can take some time, for the following reasons.

When a database instance crashes, and the physical node and operating system stay up (for example, a **shutdown abort** is performed), the termination of the process causes the operating system to clean up the TCP connection used for the SQL*Net connection. Part of this cleanup activity is to send a message to the other end of the TCP connection to inform it that the connection is being dropped. This results in an immediate "**ORA-03113: end-of-file on communication channel**" error being signalled to the user. If the operating system is not available to perform this cleanup, things are a little messier...

To test this, I wrote a small piece of Java code that connects to the database, runs a **SELECT * FROM DUAL** every 100ms. Upon an error, it exits reporting the elapsed time since it submitted the SQL that failed. Here's the results:

Using VIP:

```
morlej@pikachu:~/src/scaleabilities/tests/db> java -classpath /opt/oracle/10201/
jdbc/lib/classes12.jar:. failoverTimer jdbc:oracle:thin:@rac2-vip.sa.local:1521/
RACDB myuser myuser
Running starts at 1150305178072
Got 17002, delay=104266 ms
morlej@pikachu:~/src/scaleabilities/tests/db>
```

Using physical IP:

```
morlej@pikachu:~/src/scaleabilities/tests/db> java -classpath /opt/oracle/10201/
jdbc/lib/classes12.jar:. failoverTimer jdbc:oracle:thin:@rac2.sa.local:1521/
RACDB myuser myuser
Running starts at 1150305173253
Got 17002, delay=928681 ms
morlej@pikachu:~/src/scaleabilities/tests/db>
```

Notice the difference in delay times. The one using the VIP is lower because, when the VIP is assumed by another node, CRS sends out what is called a 'Gratuitous ARP', which is a network message declaring that the MAC address for (in this case) **rac2-vip** is now different. This causes the Operating System at the other end of the connection to drop the connection, which can then be detected by the application and dealt with accordingly: At least it knows the connection is no longer valid.

The reason that it still took 104s[1] is that there is an intentional delay in the detection of failure in a CRS cluster (or any cluster, for that matter). This delay is there to protect against false-positives; the clusterware is making *really* sure that the other node has died before instigating recovery procedures. This failure detection process is, and always has been, the bain of every cluster. On the one hand it is highly desirable to detect failures quickly so that:

- the errant node can be frozen out of the cluster, to protect against unsynchronised disk writes

1. The 104s is comprised in this case of 60s for CSS node death detection, 3s state recovery of OCR cache, 24s delay before VIP startup commenced on surviving node, and 17s to start VIP and send gratuitous ARP.

- the cluster can failover quickly, with the minimum of upset to the end user

However, doing things *too quickly* exposes the cluster to a false-positive - the node is just too busy for a few seconds to check in with its peers, but is otherwise healthy. In this case, it would be very bad indeed to freeze the node out of the cluster and to failover its services, as noted earlier.

In CRS, the default detection time is 60 checkins, at 1s intervals. So it takes a minimum of 60s before the cluster can failover a node's VIP to the other nodes. This can be reduced in your configuration, but please do watch out for those false-positives.

In the case of the physical IP connection, the failover took 928s, or about 15 minutes. This is the time that the Operating System is configured to wait before tearing down a non-responsive TCP connection. On Linux, it appears to be a function of the **`net.ipv4.tcp_retries2`** kernel parameter, which is set to 15 on my machines:

```
net.ipv4.tcp_retries2 = 15
```

Note: Changing this parameter affects **all** TCP connections from the machine. That means it affects your SSH connections and everything, so you might not want to be changing this.

This is not a linear parameter; when I set this to 10, the timeout went down to 5 minutes, not 10 minutes. According to various documents on the web, this parameter is linked to the Retransmission Timeout in the TCP RFC (793), section 3.7 Data Communication.

Anyway, it makes no sense *not* to use the VIP, especially as failover detection is then all linked to when CRS determines it rather than when a TCP connection happens to fail. For example, the TCP connection could be configured to timeout before CRS has confirmed the failure, but would then have nowhere else to connect to for the desired service. It's better to wait for everything to be synchronised on failover.

PHYSICAL FAILOVER

Now that we have detected the failure, we must decide how to handle the actual physical failover of the connection. There are a couple of approaches available to achieve the connection migration portion:

- Manually - detect the failure (manually or via one of the techniques below) and reconnect to a surviving instance
- Automatic - Transparent Application Failover (TAF) automatically moves connection to a surviving instance, or Fast Connection Failover

Transparent Application Failover. Transparent Application Failover (TAF) was designed as the 'plug and play' method of failover for Oracle RAC and Dataguard environments. The idea of TAF is that in the event of an instance crash (or other failure causing the instance to become unavailable), the user session will automatically failover to a surviving instance.

TAF has one serious limitation which stops it being a one-stop shop for connection failover - it does not support transactions at all. So, any DML submitted to the instance that just failed will need to be rolled back and *not replayed when connected to the surviving instance.* This, to me, makes it unsuitable for all but the most special case, that of a read-only application. For such an application, TAF could work well, in either of the following modes:

- SELECT failover
- SESSION failover

Both these modes are similar in that they both migrate the user session over to one of the surviving nodes. The difference with SELECT failover is that Oracle also restarts the current query from the same SCN as the initial request, and moves the row offset to the last retrieved point. To test this, I setup the following TNS entry on my client:

```
RACTAF =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = rac1-vip.sa.local)(PORT = 1521))
    (ADDRESS = (PROTOCOL = TCP)(HOST = rac2-vip.sa.local)(PORT = 1521))
    (ADDRESS = (PROTOCOL = TCP)(HOST = rac3-vip.sa.local)(PORT = 1521))
    (FAILOVER=on)
    (LOAD_BALANCE = yes)
    (CONNECT_DATA =
      (SERVER = DEDICATED)
      (SERVICE_NAME = RACDB)
      (FAILOVER_MODE = (TYPE=SELECT)(METHOD=BASIC))
    )
  )
```

Then I created another Java program, based on the last one, that spawns two hundred connections to the database and runs queries against the database. When connected, the distribution looked like this:

```
   INST_ID   COUNT(*)
---------- ----------
         1         67
         2         67
         3         66
```

I then shutdown INST3 as follows:

```
srvctl stop instance -d RACDB -i INST3 -o immediate
```

After this, my Java program was still reporting 200 healthy sessions, but the distribution looked like this:

```
   INST_ID   COUNT(*)
---------- ----------
         1        100
         2        100
```

Not bad, it seemed to work OK. Just to make sure this wasn't a fluke, I restarted INST3, reconnected the sessions, then pulled the power cable out of RAC3. Again, the failover was successful. However, as the systems out in the field are running customer applications, it is very possible that the applications in question were **not** just performing **`SELECT`** statements as declared.

The next logical test is to have some queries doing more substantial work than the comparitively atomic **`select * from dual`**!

For the next test, I created a small table, connected a single user using TAF from a client machine, and performed a simple single-row update:

```
SQL> update dmltable set a=a where a=1024;

1 row updated.
```

At this point we have an uncommitted transaction against a shortly to be failed node. After a shutdown abort of the instance this user was connected to, any simple statement received an error:

```
SQL> select * from dual;
select * from dual
*
ERROR at line 1:
ORA-25402: transaction must roll back
```

It’s worth pointing out here that the rollback is not necessary from a consistency perspective - to the database this failure is handled in just the same way as any other uncommitted transaction at instance failure: It is recovered where necessary and rolled back. However, this error is emitted by Oracle to provide the application with no possibility to proceed with a logically corrupt transaction. The application has precisely one option here: rollback the transaction. After that point, both Oracle and the application have an identical view of the transaction - the previous update is no longer applied to the database.

This time around, let’s kill the instance in the middle of a DML operation:

```
SQL> update dmltable set a=a;
update dmltable set a=a
*
ERROR at line 1:
ORA-25408: can not safely replay call
```

The reason that the call cannot be replayed is that DML is not supported by TAF - the call must be reissued. In this case, though, the application never received a successful response from the database, and so it is very clear that the DML operation must be reissued. If the TAF failover policy was ‘SESSION’ instead of the ‘SELECT’ policy used here, an ORA-25408 error would be raised even for a simple SELECT statement.

In addition to the lack of support for DML, TAF has the following restrictions:

- ALTER SESSION commands are no replayed on the failover instance. This means that NLS date formats and such could be incorrect after a failover.
- Package state cannot be relied upon - the package state is stored in the local instance. This extends to packages such as DBMS_OUTPUT that rely on a memory buffer to store lines of output.

While these restrictions remain, the failover can hardly be considered ‘transparent’ - application changes must be made for the vast majority of applications in order for the *application* to survive such a failover. However, TAF is typically the reason that most RAC users believe they have a fault-tolerant application architecture.

Fast Connection Failover. On the basis that TAF is not very satisfactory, other methods must be used to achieve the kind of failover that would be expected of a proper HA system. For applications that use Java connection pools, most commonly three-tier J2EE applications, Fast Connection Failover (FCF) provides a good starting point with very little effort on the part of the coder.

The core of FCF is the Fast Application Notification (FAN) event facility introduced with Oracle 10g. This is a formally defined mechanism by which the Oracle cluster can communicate with interested parties about state changes in the cluster. The events themselves are raised on a service or node level, and are communicated using the Oracle Notification Service (ONS). This is the part of CRS that communicates FAN events to those that want to know. These FAN events differ from other types of failure detection in that they are *logical* rather than *physical* and that they are pro-

duced on a subscribe and publish basis rather than the more traditional polling methods.

One of the great things that I found with this is that it is possible to write custom notification scripts and put them in **`$ORA_CRS_HOME/racg/usrco`***, which are then called by CRS whenever a state change occurs. In my case, I found this simplistic script* ***very*** *useful in determining what is happening from a single viewpoint:*

```
#!/bin/ksh
logger -p local6.info $*
```

All this is doing is passing the messages from ONS straight into the **`syslog`** *infrastructure of Linux. I configured the syslog service on each node to pass all messages received on the* **`local6.info`** *priority directly to my external syslog server, which in turn filtered all these messages to a unique file. This file can then just be tailed in the normal way for a single view of cluster state:*

```
Jun 16 12:57:32 rac3 logger: NODE VERSION=1.0 host=rac2 incarn=22 status=nodedown reason=member_leave times-
tamp=16-Jun-2006 12:57:32
Jun 16 12:57:34 rac1 logger: NODE VERSION=1.0 host=rac2 incarn=22 status=nodedown reason=member_leave times-
tamp=16-Jun-2006 12:57:32
Jun 16 12:57:42 rac3 logger: SERVICE VERSION=1.0 service=myserv2 database=RACDB instance= host=rac3 status=up rea-
son=unknown timestamp=16-Jun-2006 12:57:42
Jun 16 12:57:44 rac1 logger: DATABASE VERSION=1.0 service=RACDB database=RACDB instance= host=rac1 status=up rea-
son=unknown timestamp=16-Jun-2006 12:57:44
Jun 16 13:02:07 rac2 logger: ASM VERSION=1.0 service= database= instance=ASM2 host=rac2 status=up reason=boot
timestamp=16-Jun-2006 13:02:07
Jun 16 13:02:53 rac2 logger: INSTANCE VERSION=1.0 service=RACDB database=RACDB instance=INST2 host=rac2 status=up
reason=autostart timestamp=16-Jun-2006 13:02:52
```

As can be seen in the above breakout, it is now possible to receive notification of any state change within CRS. These messages can also be digested by subscribing applications, allowing the application to make informed decisions about where and when to connect to the database. It is also the cornerstone of Fast Connection Failover.

FCF uses the FAN events in conjunction with a local ONS daemon and the OracleDataSource connection pool (or Connection Cache) to produce an automagically managed connection pool that is responsive to FAN events. Let's look at what all that means.

The first component in the chain is the local ONS daemon. This is a standard ONS daemon like the one running on the database server, but this one resides on the application server itself. This ONS daemon acts as a multiplexer for incoming FAN events - the database server only has to be concerned with the address/port of that daemon when communicating the FAN events.

The Java code within the Oracle JDBC driver (both 'thick' and 'thin' versions), specifically the code underneath the OracleDataSource class, then communicates with the local ONS daemon without needing to be aware of the database server ONS configuration.

Once configured correctly (see [OOWFAN04] for the gory details), the OracleDataSource (ODS from now on) will respond to both UP and DOWN events through the ONS communication mechanism. It is also now possible (from 10.2, I believe) to eliminate the requirement for the local ONS daemon, and to set the names of the RAC nodes manually by using the following JDBC syntax:

```
OracleDataSource.setONSConfiguration("nodes=machine1:port,machine2:port)
```

So what can we expect, now that we have Oracle managing our connection pool and the failover within? Unfortunately, the answer is: Not much. Here's why.

The standard way to use a connection pool is to 'get' a connection from the pool, effectively borrowing it from the available connections, then execute some SQL, then give back the connection to the pool. It is the responsibility of the application

logic to trap any errors, such as primary key violations, in exception handlers, and to deal with it as appropriate. The way FCF handles service failures is to signal an ORA-17008 (Closed Connection) error SQLException back to the developer. What FCF gives the developer is just yet another error to trap, in much the same way as TAF, but potentially not as clean. Consider this comparison of TAF and FCF:

Situation	How TAF Handles It	How FCF Handles It
Loss of current connection	Fails over the connection to a surviving node	Expires the connection from the connection pool
Use of a newly failed connection: no active TX	Connection operates without issue	ORA-17008
Use of a newly failed connection: active TX in flight at time of failure	ORA-25408 (can not safely replay call)	ORA-17008
Use of a newly failed connection: active TX completed but uncommitted at time of failure	ORA-25402 (must rollback)	ORA-17008
Failover Time	As soon as the VIP is up and broadcasts gratuitous ARP (up to 104s on my test system)	As soon as the DOWN event is triggered (60+ seconds on my test system)
Node returns to service	Needs connection pool support to add connections back to node	Can automatically add connections back to node

So, in summary, FCF gives a small improvement in failover time compared to a VIP, and some handling for node UP events. It still requires logic in the body of the application (outside the connection pool) to handle connection failover.

Failover Summary

Connection failover is a complex issue. Given the limitations of both TAF and FCF, both of which will ultimately require careful application support, it is probably a more stable solution to just bite the bullet and write the required code to support the availability needs of the application.

The simplest approach may be to rely solely on the VIP/gratuitous ARP combination to detect the fault. Though slower than using FCF, there is no requirement for configuration support using this method. TAF could be used, along with its more meaningful errors, and at least SELECT statements would not need any special handling, nor would connections need to be re-established manually.

Alternatively, FCF could be adopted. However, careful support needs building into the application, probably in terms of building some subclasses of the standard JDBC primitives to track the state of (notably) DML transactions, plus the availability of session state. For example, subclass the JDBC primitive classes to trap for 17008 errors and to trigger the required handler for that state. The handling might look something like this:

- discard the connection back to the pool and retrieve a valid one

- replay all “ALTER SESSION” commands
- replay all DML statements since last commit
- replay the last requested SQL that triggered the error and return the anticipated result object back to the caller

Of course, it’s not quite that simple. Consider the situation where a node fails, thus implicitly rolling back any uncommitted DML, and a user on another node performs an incompatible update on another node before the FCF has alerted the original session of the failure. There’s a good chance of this, given the time windows involved, by the way. For example:

- I update a price in a stock table, where the stock code is 90125 (1 row updated)
- My node fails
- Another user, on another node, decides to change the stock code for this item
- I finally start replaying my previous transaction, but now the row does not exist (0 rows updated)

If I still had that row locked, like I did before the server died, I would have been successful. But that sneaky guy changed the stock code on me while I was in limbo, and now I’m trying to safely replay the DML back in an automated handler. Well guess what? There will always be a risk that such an automated handler makes the wrong decision. Even if it checked that the number of rows updated were the same as the previous attempt to execute it, what if the sneaky guy on the other node had made a **new** stock item with a code of 90125? It would still be the same number of rows updated, and there is no way that the handler can tell the difference.

So, the handling of failures is completely specific to the situation in which it occurs. Don’t worry too much about your queries (unless they are in the middle of some DML), but any parts of the data layer that result in modification to data need to be written with failure in mind.

Services

No paper on RAC connection management would be complete without a mention of services. Services have been around for a little while now, but as of Oracle 10gR2, the integration of services is very comprehensive. No longer constrained to being just a connection convenience, the principle of services is now applied to the collection of statistics, to trace file gathering, and to resource management.

From a TNS perspective, services can be viewed as an extra layer on top of all the load balancing and failover topics that we looked at earlier in this paper. Services allow operational control of the node placement of physical connections both initially and after failures.

Due to the depth of coverage of services, I will defer this study until version two of this paper, lest this becomes a perpetually unfinished work!

Connection Management Summary

Connection management in a RAC environment is a complex affair if the desired availability result is to be achieved. With careful configuration, reliable load balancing is possible in a manner that is useful to the application - both from a workload balancing and from a workload partitioning perspectives. Connection failover, though, is far more complex than the load balancing. Though there are many facilities in place to try and detect the failure more quickly, the handling of those failures quickly falls back into the domain of the application developer.

References

[OOWFAN04] - How To Build End-to-End Recovery and Workload Balancing for you Applications, Oracle Open World 2004: *Barb Lundhild, Carol Colrain, Troy Anthony, Daniel Semler, Rajkumar Irudiyaraj*

About The Author

James Morle is the founder of Scale Abilities Ltd, a UK-based consulting practice and services company focused on Oracle and the stickier parts of the associated platform engineering such as clusters and complex SAN issues. With 15 years of Oracle experience, mostly all with systems with various degrees of scaling insanity, James has a more than a passing interest in knowing how things work for real. As a founder member of the OakTable Network (www.oaktable.net) he can frequently be found at various events locked in conversation with other similarly disturbed individuals. As a founder of the Baarf Party (www.baarf.com) he can never be found explaining why RAID-F is bad. Ever again.