Best Practices for Using XA with RAC

This document lists considerations when using XA in a RAC environment, and provides the recommended solutions for both Oracle9i and Oracle Database 10g.

Split-branches of a distributed transaction

A split transaction is a distributed transaction that spans more than one instance of a RAC database. This implies different branches of the same distributed transaction are located on different instances of a RAC database. Two situations can occur.

During normal operation: neither branch can see changes made by the other branch. This can cause row-level lock conflicts amongst these branches leading to ORA-2049 errors (timeout waiting for distributed lock).

During recovery operation: failures can occur during two-phase commit (2PC). Sometimes 2PC requires its own connection to the database (e.g. an abort). In such cases, a 2PC operation may be attempted on a transaction branch at an instance where that branch does not exist, causing ORA-24756. This in-turn leaves the branch hanging as an active branch to be cleaned up by PMON. While the branch is active, it still holds row-level locks on all rows that it modified. Similarly, if the 2PC operation that failed is a commit, then in-doubt transactions can remain in the database. This can cause ORA-1591 errors when another transaction attempts to access data that has been modified by the in-doubt transaction.

Solution

To avoid these conditions all related work from the same 'business' transaction in an XA environment must be directed to a single RAC instance.

This can be accomplished by creating database services[1] that each run on one instance at a time. Refer to these services, as 'singleton' services because they have 1 preferred instance defined so they are only ever active on one instance in the cluster. It is important that we never allow these services to be active on more than one instance at a time in any situation.. This means that when a service is relocated to another instance – for example, during switchover and fallback - ALL sessions must be disconnected before any transactions use the new instance. Otherwise, new sessions for a single business transaction could start up on the new instance, while old sessions are still running on the old instance.

The way to use singleton services differs, if one is using Oracle Database 10g or Oracle9i.

[1] For more information on Services, read the RAC Administration and Deployment Guide Chapter 6 (B14197-03)

Oracle Database 10g - Services provide Automatic Workload Management (AWM)

TP monitors and application servers operate on services. In Oracle Database 10g, the notion of a service with Automatic Workload Management can be used to direct work to specific instances, and to automatically redirect this work to other instances should the active instance(s) fail.

Services are mapped to application functions, or application data ranges or hash values. Each of these divisions of work can be used to provide instance affinity for the business transaction. For example bank branches can be grouped together and mapped to services. The mapping from clients or application functions to service is maintained in BC4J for Oracle AS, in the bulletin board for Tuxedo, in LDAP for MTS and so on.

Start by creating many more services than instances – say 16 services. By using many more services than instances, instances and nodes can then be added and removed, and the system remains balanced. Give each service a cardinality of one – that is, a service runs on exactly one instance at a time. With Oracle Database 10g, high availability and preferred placement are managed automatically. Since the cardinality for each service is one (only 1 preferred instance), there are no issues for XA. There are also no restrictions regarding the use of one or more mid-tier servers by a single business transaction.

To ensure that sessions are never active on more than one instance at a time, when moving a service, use *srvctl stop service disconnect.* This will disconnect all the existing sessions before the service is started on another instance. To ensure an operator does not forget to use the 'disconnect' clause, create a FAN (Fast Application Notification) callout[2] that always disconnects the existing sessions when the service stops.

Oracle Database 10g Release 2 – Distributed Transaction Processing Services

Specific to XA, Oracle Database10g Release 2 provides services that are optimized to support Distributed Transaction Processing (DTP). This feature optimizes XA transaction recovery. All the in-flight prepared transactions belonging to a DTP service of the failed instance are pushed to the disk table before the DTP service is re-started on the any of the surviving instances. In-flight transaction prepare info is written to the undo area and only if the transaction fails, this information is pushed into the appropriate system table to allow external recovery to take place. During normal operations, this push is done lazily. But as soon as the DTP service fails over users expect the failed (prepared) distributed transaction info on the system table to able to recover the transaction. Thus we make sure all the required information is propagated to the system tables before starting

[2] Sample callout can be found on OTN
http://www.oracle.com/technology/sample_code/products/rac/index.html

the DTP service on one of the surviving instances. This ensures that all the in-doubt transactions will be reported by xa_recover() and a following xa_commit() or, xa_rollback() will be able to complete the in-doubt transactions. This feature also guarantees that all branches of single global distributed transactions are maintained at a single RAC instance.

This mechanism optimizes the XA recovery by recovering the transactions once, at speed, as part of the transaction recovery code. The XA/DTP feature also guarantees correctness by ensuring that all branches of a transaction are maintained at the same database instance. This includes switch over and failover of services for planned and unplanned outages. The feature is only available for services declared as DTP.

Tip: When creating Services, create many more services than instances to allow ease of load balancing.

Example Deployment using BEA XA Tuxedo Service Groups with Oracle 10g Services:

Step 1. Create Mid-Tier Service Groups
Create several Tuxedo service groups comprising the Tuxedo services that transactions span. Each service group is a clone of the others. The groups can be replicated at each mid-tier server, or can use backup groups in the same domain.

Example of Replicated Server Groups

midtier server 1:
Service Group A comprises functions (services) {fa, fb, ... fg}
Service Group B comprises functions (services) {fa, fb, ... fg}
Service Group C comprises functions (services) {fa, fb, ... fg}

midtier server 2:

Service Group A comprises functions (services) {fa, fb, ... fg}
Service Group B comprises functions (services) {fa, fb, ... fg}
Service Group C comprises functions (services) {fa, fb, ... fg}

Example of Backup Server Groups

midtier server 1:
Service Group A_master comprises functions (services) {fa, fb, ... fg}
Service Group B_master comprises functions (services) {fa, fb, ... fg}
Service Group C_master comprises functions (services) {fa, fb, ... fg}

midtier server 2:

Service Group A_backup comprises functions (services) {fa, fb, ... fg}
Service Group B_backup comprises functions (services) {fa, fb, ... fg}
Service Group C_backup comprises functions (services) {fa, fb, ... fg}

Step 2. Configure Oracle 10g Services for XA

Create Oracle Database 10g Services that map to each master mid-tier service group. When using XA, the Oracle side Services must have cardinality one (that is, one preferred instance). This can easily be completed by setting the parameter DTP=>true on the service (using DBMS_SERVICE or Enterprise Manager Cluster Managed Services page). DTP services are optimized for XA recovery as well as guaranteeing single cardinality.

Example:Using XA with Tuxedo Service Groups and Oracle 10g Services

With BEA Service Group, and also BEA Domains, and Oracle Database 10g Services, high availability and load balancing are provided out of the box. For example, you can use gold, silver, and bronze services and manage priority via resource manager. Likewise you can classify groups of users into services and measure and manage their performance. The usage of services goes beyond XA.

In this example, create services: Service A, Service B, Service C. Note that each service uses exactly one preferred instance.

```
srvctl create service -d MYDB -s ServiceA - r MyDB1 -a MyDB2, MyDB3
srvctl create service -d MYDB -s ServiceB - r MyDB2 -a MyDB1, MyDB3
srvctl create service -d MYDB -s ServiceC - r MyDB3 -a MyDB1, MyDB2
```

Starting each service leads to the DBMS automatically discovering the service.

```
srvctl start service -d MYDB -s ServiceA
srvctl start service -d MYDB -s ServiceB
srvctl start service -d MYDB -s ServiceC
```

Modify each service to support XA.

```
execute dbms_service.modify_service (service_name => 'ServiceA' , dtp => true);
execute dbms_service.modify_service (service_name => 'ServiceB', dtp => true);
execute dbms_service.modify_service (service_name => 'ServiceC', dtp => true);
```

Step 3. Assign each BEA Service Group to each Oracle Service

Each mid-tier Service Group uses an Oracle Service. Since the Oracle Services are "DTP", within transaction affinity, high availability, and fast recovery are automatic.

Example of Replicated Server Groups

Service Group A - uses Service A
Service Group B - uses Service B
Service Group C - uses Service C

Example of Backup Server Groups
Service Group A_master, A_backup - use Service A
Service Group B_master, B_backup -use Service B
Service Group C_master, C_backup -use Service C

Step 4. Load Balancing
Clients are balanced across the Service Groups (or Domains) via keys that are appropriate to the application. The more service groups and services, the more effective and more flexible the system. The service groups and services are a unit for high availability, manageability, and workload management.

The choices for the load balancing keys are many and varied. The method chosen should suite the application. This can be done using separate physical domains with clients divided across these domains, or by using data dependent routing within the same Tuxedo domain. It's important to note that for different transactions, different keys can be used. The only rule is all branches of the same transaction must use the same service group and service.
Examples are:

1. Data dependent ranges using Tuxedo's very powerful Data Dependent Routing. This is the best method to use because it facilitates workload management and planned outages, as well as load balancing. Examples of keys are easy such as Client-Logon-Id, application focused such as flight number, destination code, and physical such as transaction id.

2. Hashed – Hash is a new algorithm proposed by BEA. The hash algorithm should hash to the DTP services (not instances), maintaining the XA integrity and correctness, and recovery speed. The input to the hash scheme can be opaque such as transaction id, or logical, similar to data dependent ranges, using client-logon-id, flight number and so on .

3. Domains – clients are routed to a parent domain. Within this domain, clients are always routed to the same child domain. This routing can be based on CPU load or a data dependent key.

   Example of Replicated Server Groups

   Transaction t1 uses Service Group A, Service A
   Transaction t2 uses Service Group B, Service B
   Transaction t3 uses Service Group C, Service C

   Example of Backup Server Groups

   ```
   Transaction t1 uses Service Group A_master with backup, A_backup
   ```

```
Transaction t2 uses Service Group B_master with backup, B_backup
Transaction t3 uses Service Group C_master with backup, C_backup
```

Data dependent routing and hashed methods provide the most flexibility for scaling and load balancing, as well as planned outages.

For more information on Services, refer to *Oracle Clusterware and RAC Administration and Deployment Guide, 10g Release 2, Chapter 6*.

BEA WebLogic Server 8.1 SP4 MultiPool Patch and Oracle RAC 10g

WebLogic Server 8.1 SP4 now provides a patch that enables BEA WebLogic Server 8.1 SP4 MultiPools to be used in conjunction with Oracle RAC 10g. With this patch, Weblogic users can take advantage of connection pool failover and load balancing for applications using XA transactions. BEA recommends that you apply this patch to your system if you are using WebLogic Server 8.1 SP4 with Oracle RAC 10g and are using XA transactions. This patch is compatible with non-XA usage, and incorporates support previously delivered in the WebLogic Platform 8.1 SP4 patch for Oracle9i RAC driver-level load balancing support.
This WebLogic Server 8.1 SP4 support patch is supported by WebLogic Server 8.1 SP4 only; it is not supported by other WebLogic Platform products. The patch is available at the following URL: http://dev2dev.bea.com/wlserver/patch/wls81sp4_MP_OracleRAC_patch.html

For more information on using WebLogic Multipools with RAC, go to the BEA Website: http://e-docs.bea.com/wls/docs81/jdbc/oracle_rac.html or http://e-docs.bea.com/wls/docs92/jdbc_admin/oracle_rac.html

Note: BEA recommends turning off server-side connection load balancing in the above document (and the more recent document for WebLogic 9). Please note that this is not necessary nor recommended when using a DTP service with Oracle RAC 10g Release 2 or when using a service with one preferred instance with Oracle RAC 10g Release 1.

Oracle 9.2 Custom Solution

Unfortunately, Automatic Workload Management is not available in Oracle 9.2 to simplify the direction and redirection of a service to specific nodes. So, the solution requires custom code. In Oracle 9.2, service refers the service name descriptor as used in TNS names, and to the service_names parameter for accepting connections to that service.

Creation of singleton Oracle database services

Assume that there are two RAC instances I1 and I2. Create two singleton database services apps_service1 and apps_service2. In order to be able to dynamically shift service across RAC nodes do not set the service_name parameter in the database initialization file for any of these instances. Instead, set the service name for an instance using the "ALTER SYSTEM SET SERVICE_NAME=<service name>" command.

Creation of BEA Tuxedo service groups

In order to route the mid-tier clients across the various RAC nodes, create two or more BEA Tuxedo service groups. Each of these Tuxedo service groups maps to one and only one Oracle database service. The relationship between the Tuxedo service groups and the Oracle RAC services can be one-to-one or many-to-one, so long as each business transaction routes to one RAC instance. The mid-tier directs the clients to the Tuxedo service groups, for example based on application function, data values, and importance.

Setting up TNSNAMES.ORA entries for each BEA Tuxedo service group

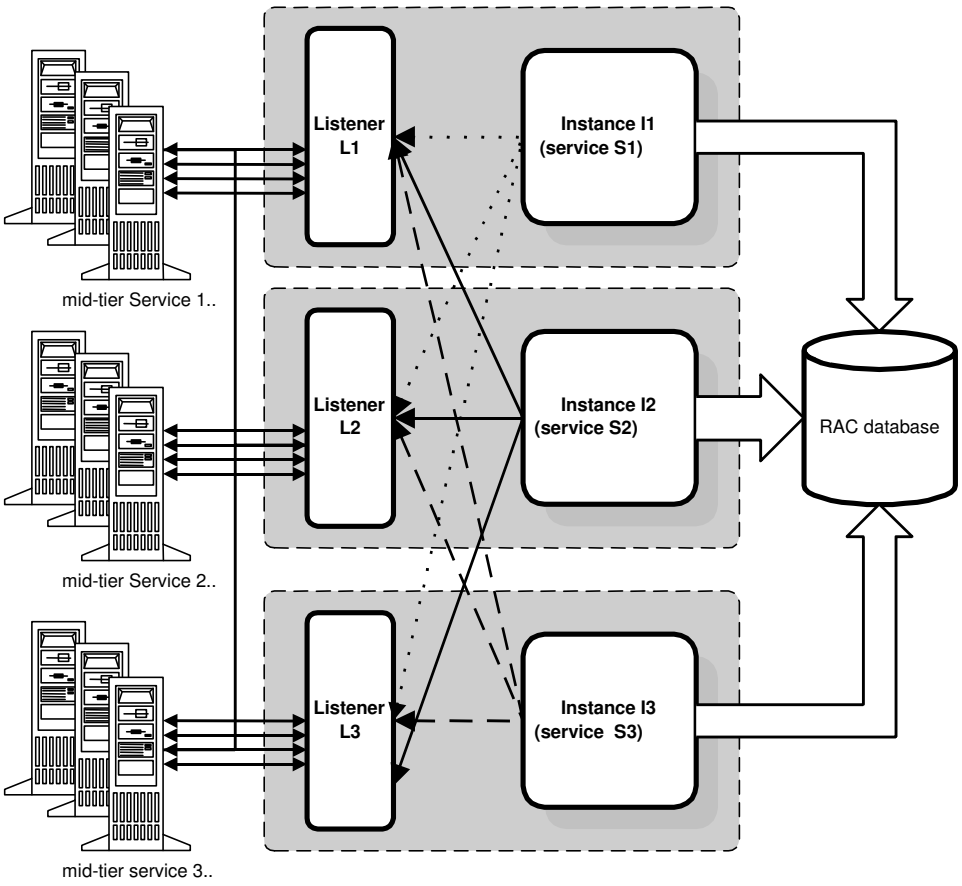
A given Tuxedo service group uses its own TNSNAMES.ORA entry that maps to a single Oracle RAC database service name.

SQL*Net alias for Tuxedo service group 1

```
oracle_rac1=
  (DESCRIPTION =
      (ADDRESS_LIST =
          (ADDRESS = (PROTOCOL=TCP)(HOST = racnode1)(PORT = 1521))
          (ADDRESS = (PROTOCOL=TCP)(HOST = racnode2)(PORT = 1521))
      )
     (CONNECT_DATA = (SERVICE= apps_service1))
     (FAIL_OVER = ON)
  )
```

SQL*Net alias for Tuxedo service group 2

```
oracle_rac2=
  (DESCRIPTION =
      (ADDRESS_LIST =
          (ADDRESS = (PROTOCOL=TCP)(HOST = racnode1)(PORT = 1521))
          (ADDRESS = (PROTOCOL=TCP)(HOST = racnode2)(PORT = 1521))
      )
     (CONNECT_DATA = (SERVICE= apps_service2))
     (FAIL_OVER = ON)
  )
```



RAC using services for instance affinity

Automating Service Failover and Fallback in Oracle 9.2

The requirements for the custom solution are:

1. Failover the services whenever an instance or node goes down, as detected by a reconfiguration change.
2. Fallback the services whenever an instance resumes.
3. Do not allow the same service to be published in two places.
4. Ignore false reconfiguration events and time-outs.

Design

Write a simple PL/SQL function (or C program) that monitors one instance from another. When that instance shuts down, takeover the service_name that the instance supported.

Use dbms_system.wait_for_event(‘ges reconfiguration to start’’, [parameter 1], [parameter 2]); to detect and act on reconfiguration events. See DBMS_SYSTEM.

1. Create a job or procedure on each node, connected to the running instance that waits for the other instances to leave or join the cluster. For example, this can be an init.d process, a cron process, or a dbms_job.
2. When the procedure starts, and when the function dbms_system. wait_for_event, times-out or completes, select the status of the instance that is being monitored: *select status from v$instance where instance_name ='….';*
3. If the status of the instance being monitored has changed since the last check on status, take the action in table 1. For example, if the instance was OPEN and is now CLOSED, take over the service_name (see PL/SQL below). If, the instance was OPEN and is now still OPEN, take no action. If the instance was MOUNTED and is now OPEN – also take no action.
4. After completing the action in table 1, save the status of the instance being monitored and block again waiting for reconfiguration. This saved status will be compared when the wait event returns.
5. Continuously repeat steps 2, 3 and 4.

Starting the service at an instance is manual because all sessions on the hosting instance must be complete. When starting a service, check that the service is not registered at another instance. This check can be added to the PL/SQL code provided.

Exactly one instance can take over a service. In table 1 - the instance with lowest instance number takes over the service. When using a two node RAC, this extra check is not required.

Table 1. Procedure to monitor instance A from another instance.
Status and Instance_Number are available in v$instance. When wait_for_event returns compare the last status of the instance to the current status of the instance.

Last status for instance A	Current status for instance A	Action
Open, Mount	Open, Mount	Remember the new status in v$instance. Go back to waiting for reconfiguration
Closed (or nomount or restricted)	Closed (or nomount or restricted)	Remember the new status. Go back to waiting for reconfiguration
Open, Mount	Closed (or nomount or restricted)	/* Take over the failed instance's service */ (If this is the lowest number instance) register the failed instance's service at this instance. Remember the new status. Go back to waiting for reconfiguration
Closed (or nomount or restricted)	Open, Mount	/* Release the hosted service(s) */ Deregister the service locally, and quiesce sessions using this service; to avoid split branches.

		Go back to waiting for reconfiguration. Remember the new status. * Manually register the service at the newly opened instance *

Utility PL/SQL package

The following PL/SQL package (with some modifications) can be used to allow an instance to provide an additional service or to stop an instance from providing a specified service:

```
create or replace package service_utility
as
  /* register this instance as providing the service new_service_name */
  procedure register_service(service_name varchar2);

  /* register this instance as no longer providing the service
old_service_name */
  procedure unregister_service(service_name varchar2);

  /* cleanup all sessions in this instance for connections to
cleanup_service_name */
  procedure cleanup_service(service_name varchar2);
end service_utility;
/

create or replace package body service_utility
as
  procedure register_service (service_name varchar2)
  is
     old_service_names_list               varchar2(2000);
     new_service_names_list               varchar2(2000);
     alter_statement                      varchar2(2000);
     new_service_name                     varchar2(128);
  begin
    -- retrieve current service names registered
    select value into old_service_names_list
          from v$parameter
          where name = 'service_names';
    -- check if we are already registered, if so raise user exception
    old_service_names_list := lower(old_service_names_list);
    new_service_name := lower(service_name);
    dbms_output.put_line('current services: ' ||ld_service_names_list);
    if instr(old_service_names_list, new_service_name) <> 0
    then
       raise_application_error(-20000,
      'instance already registered as providing service: ' || new_service_name);
    end if;
    -- build new list of service names
    new_service_names_list := old_service_names_list || ',' ||  new_service_name;
    dbms_output.put_line('new services: ' || new_service_names_list);
    -- construct the ALTER SYSTEM SET SERVICE_NAMES stmt
    -- and execute the statement
    alter_statement := 'ALTER SYSTEM SET SERVICE_NAMES = ''' ||
                        new_service_names_list || '''';
    execute immediate alter_statement;
  end;

  procedure unregister_service(service_name varchar2)
```

```
  is
     old_service_names_list  varchar2(2000);
     new_service_names_list  varchar2(2000);
     old_service_name        varchar2(128);
     alter_statement         varchar2(2000);
  begin
    -- retrieve current service names registered
    select value into old_service_names_list
       from v$parameter
       where name = 'service_names' ;
    -- check if we are registered as this service, if not so raise user
exception
    old_service_names_list := lower(old_service_names_list);
    old_service_name := lower(service_name);
    dbms_output.put_line('current services: ' ||ld_service_names_list);
    if instr(old_service_names_list, old_service_name) = 0
    then
       raise_application_error(-20001,
           'instance not registered as providing service: ' ||
old_service_name);
    end if;
    -- build new list of service names, substitute dummy_service for
old_service_name
    new_service_names_list := replace(old_service_names_list,
                              old_service_name, 'dummy_service');
    dbms_output.put_line('new services: ' || new_service_names_list);
    -- construct the ALTER SYSTEM SET SERVICE_NAMES stmt
    -- and execute the statement
    alter_statement := 'ALTER SYSTEM SET SERVICE_NAMES = ''' ||
                       new_service_names_list || '''';
    execute immediate alter_statement;
  end;

 procedure cleanup_service(old_service_name varchar2)
  is
     type SidCurTyp is ref cursor;
     c1                SidCurTyp;
     alter_statement   varchar2(128);
     cleanup_service   varchar2(128);
     session_id        number;
     serial_num        number;
  begin
   cleanup_service := lower(old_service_name);
   dbms_output.put_line('cleaning up sessions for service: ' ||
cleanup_service);
   -- open the cursor using the argument as the bind var value
   open c1 for
         'select sid, serial# from v$session where
              lower(service_name) = :cleanup_service'
         using cleanup_service;
    -- for each session connected for the specified  service construct the
    -- ALTER SYSTEM KILL SESSION statement and execute it immediately
    loop
      fetch  c1 into session_id , serial_num;
      exit when c1%notfound;
      alter_statement := 'alter system kill session ''' ||
                         session_id || ',' || serial_num || '''';
      execute immediate alter_statement;
      dbms_output.put_line('killed session: <' ||
                           session_id || ',' || serial_num || '>');
    end loop;
    if c1%isopen then
```

```
      close c1;
    end if;
  end;
end;
/
```

Handling instance failure

When an instance (say I1) fails, the monitor program starts the services that were previously available at the failed instance.

```
SQL>set instance inst2
SQL>connect / as sysdba
SQL>execute dbms_system.dist_txn_sync();
SQL>execute service_utility.register_service("apps_service1");
SQL>exit
```

As a result of this script, any new connections to service apps_service1 are routed to instance I2. Note that any work done so far by transactions on instance I1 and continued after I1's failure on I2 will not reach completion, since the 2-phase commit of these transactions will rollback.

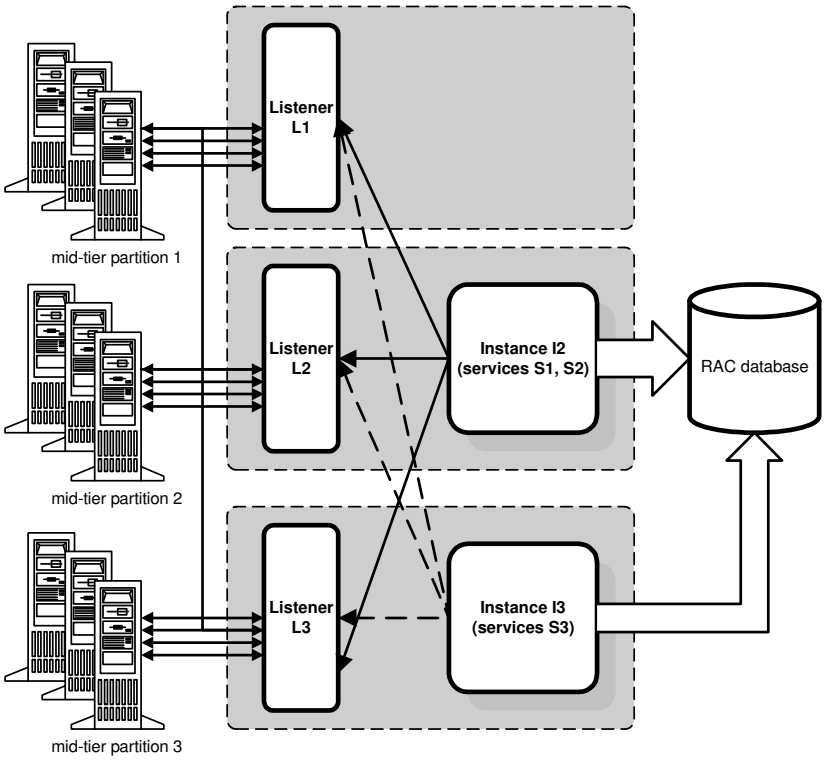


Figure 1. Shutdown of instance I1

Handling fallback i.e. when dead instance I1 comes back

When instance I1 that formerly hosted service apps_service1 comes back up deregister the service that would be provided by instance I1 from the instance currently hosting that service. Also ensure that all connections for service apps_service1 that have been routed

to the host are terminated before making the service available at instance I1. If this is not done, transactions could be split across the two instances.

```
SQL>set instance inst2
SQL>connect / as sysdba
SQL>execute
service_utility.unregister_service("apps_service1");
SQL>execute
service_utility.cleanup_service("apps_service1");
SQL>execute dbms_system.dist_txn_sync();
SQL>set instance inst1
SQL>connect / as sysdba
SQL>startup pfile=……
SQL>exit
```

Deregistering instance I2 from service apps_service1 prevents listeners from routing any more connections for that service to instance I2. In fact no new connections will be accepted for apps_service1 till we cleanup any remnants of sessions related to apps_service1 on its current provider (i.e. instance I2). The cleanup of the any existing sessions that connected to instance I2 for service apps_service1 prevents split-branches.

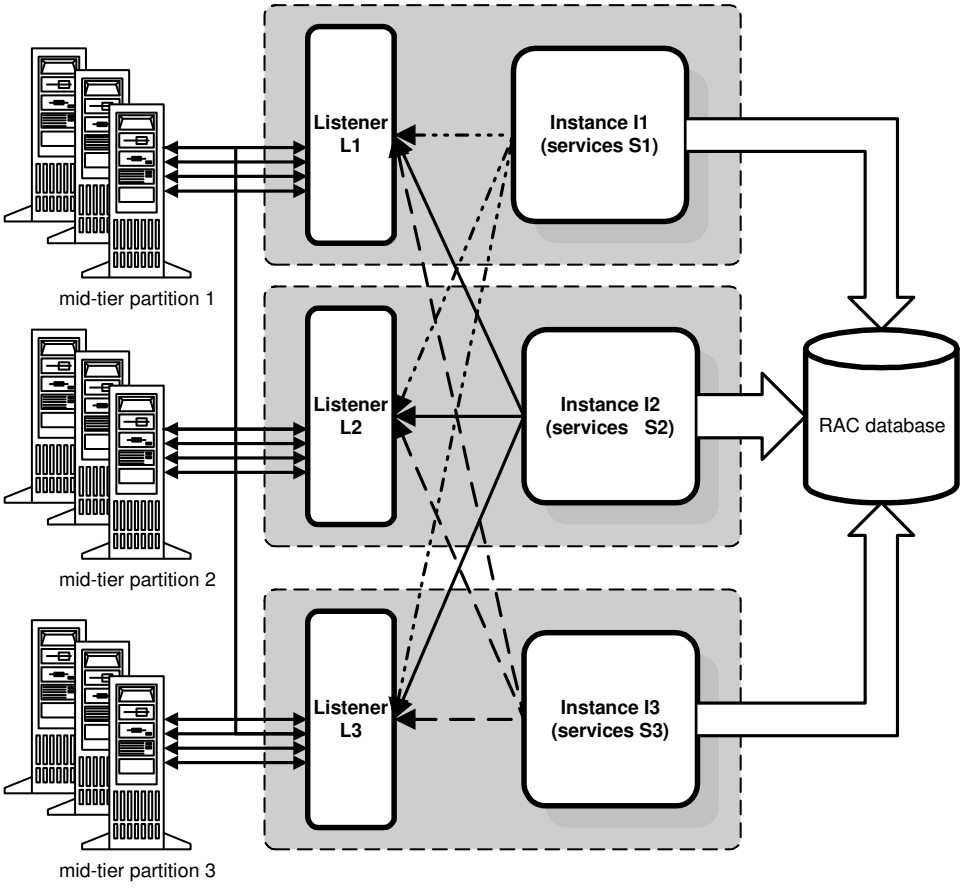


Figure 2. Fallback of instance I1

Does this solution avoid split branches?

This solution avoids split branches when all instances are alive. No connection from a mid-tier is routed to the wrong instance. Each BEA Tuxedo service group uses its own database service and at a given point in time a service can only be provided by a single instance i.e. the service's primary provider.

When an instance shuts down, it is possible for split branches to occur. In this case, all branches on the primary provider of that service are aborted (the whole instance is down). Any work performed in the split branches on hosts for that service is rolled back. Split branches can also occur in a fallback situation. In this case any work done on the backup service provider is rolled-back since we snipe all sessions on the provider for the failed service before letting the primary service provider come up. Thus any continuing work on the primary service provider post fallback can never commit.

Author: Barb Lundhild, Carol Colrain, Vivek Raja

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com