149
DBMS_PIPE

The `DBMS_PIPE` package lets two or more sessions in the same instance communicate. Oracle pipes are similar in concept to the pipes used in UNIX, but Oracle pipes are not implemented using the operating system pipe mechanisms.

This chapter contains the following topics:

DBMS_PIPE Overview

Pipe functionality has several potential applications: external service interface, independent transactions, alerters (non-transactional), debugging, and concentrator.

- External service interface: You can communicate with user-written services that are external to the RDBMS. This can be done effectively in a shared server process, so that several instances of the service are executing simultaneously. Additionally, the services are available asynchronously. The requestor of the service does not need to block a waiting reply. The requestor can check (with or without time out) at a later time. The service can be written in any of the 3GL languages that Oracle supports.
- Independent transactions: The pipe can communicate to a separate session which can perform an operation in an independent transaction (such as logging an attempted security violation detected by a trigger).
- Alerters (non-transactional): You can post another process without requiring the waiting process to poll. If an "after-row" or "after-statement" trigger were to alert an application, then the application would treat this alert as an indication that the data probably changed. The application would then read the data to get the current value. Because this is an "after" trigger, the application would want to do a "`SELECT FOR UPDATE`" to make sure it read the correct data.
- Debugging: Triggers and stored procedures can send debugging information to a pipe. Another session can keep reading out of the pipe and display it on the screen or write it to a file.
- Concentrator: This is useful for multiplexing large numbers of users over a fewer number of network connections, or improving performance by concentrating several user-transactions into one DBMS transaction.

DBMS_PIPE Security Model

To use `DBMS_PIPE` and its subprograms, you must be granted the `EXECUTE` privilege on the package. Security can further be achieved by creating pipes using the `private` parameter in the `CREATE_PIPE` function and by writing common packages that only expose particular features or pipenames to particular users or roles.

Depending upon your security requirements, you may choose to use either public pipes or private pipes, which are described in DBMS_PIPE Operational Notes.

The `DBMS_PIPE` package uses invoker's rights, meaning the operations of sending and receiving messages run in the invoker's schema.

In order to use `DBMS_PIPE` messages with Cloud Object stores, you must have the `EXECUTE` privilege on the `DBMS_PIPE` package, the `DBMS_CLOUD` package, and on the Credential Object for accessing the object store URI.

> **See Also:**
>
> - DBMS_CLOUD for information about the `DBMS_CLOUD` package and about working with cloud service credentials

DBMS_PIPE Constants

This is the maximum time to wait attempting to send or receive a message.

```
maxwait   constant integer := 86400000; /* 1000 days */
```

DBMS_PIPE Operational Notes

Except for persistent pipes, information sent through Oracle pipes is buffered in the system global area (SGA). All information in pipes is lost when the instance is shut down.

> **WARNING:**
>
> Pipes are independent of transactions. Be careful using pipes when transaction control can be affected.

The operation of `DBMS_PIPE` is considered with regard to the following topics:

- Public Pipes
- Writing and Reading Pipes
- Private Pipes
- Singleton Pipes
- Persistent Pipes

Public Pipes

You may create a public pipe either implicitly or explicitly. For *implicit* public pipes, the pipe is automatically created when it is referenced for the first time, and it disappears when it no longer contains data. Because the pipe descriptor is stored in the SGA, there is some space usage overhead until the empty pipe is aged out of the cache.

You create an *explicit* public pipe by calling the `CREATE_PIPE` function with the `private` flag set to `FALSE`. You must deallocate explicitly-created pipes by calling the `REMOVE_PIPE` function.

The domain of a public pipe is the schema in which it was created, either explicitly or implicitly.

Reading and Writing Pipes

Each public pipe works asynchronously. Any number of schema users can write to a public pipe, as long as they have `EXECUTE` permission on the `DBMS_PIPE` package, and they know the name of the public pipe. However, once buffered information is read by one user, it is emptied from the buffer, and is not available for other readers of the same pipe.

The sending session builds a message using one or more calls to the `PACK_MESSAGE` procedure. This procedure adds the message to the session's local message buffer. The information in this buffer is sent by calling the `SEND_MESSAGE` function, designating the pipe name to be used to send the message. When `SEND_MESSAGE` is called, all messages that have been stacked in the local buffer are sent.

A process that wants to receive a message calls the `RECEIVE_MESSAGE` function, designating the pipe name from which to receive the message. The process then calls the `UNPACK_MESSAGE` procedure to access each of the items in the message.

Private Pipes

You explicitly create a private pipe by calling the `CREATE_PIPE` function. Once created, the private pipe persists in shared memory until you explicitly deallocate it by calling the `REMOVE_PIPE` function. A private pipe is also deallocated when the database instance is shut down.

You cannot create a private pipe if an implicit pipe exists in memory and has the same name as the private pipe you are trying to create. In this case, `CREATE_PIPE` returns an error.

Access to a private pipe is restricted to:

- Sessions running under the same userid as the creator of the pipe
- Stored subprograms executing in the same userid privilege domain as the pipe creator
- Users connected as `SYSDBA`

An attempt by any other user to send or receive messages on the pipe, or to remove the pipe, results in an immediate error. Any attempt by another user to create a pipe with the same name also causes an error.

As with public pipes, you must first build your message using calls to `PACK_MESSAGE` before calling `SEND_MESSAGE`. Similarly, you must call `RECEIVE_MESSAGE` to retrieve the message before accessing the items in the message by calling `UNPACK_MESSAGE`.

Singleton Pipes

Singleton pipes provide the ability to cache a single message in the shared memory of the current database instance, allowing high throughput concurrent reads of a message across database sessions. In a RAC database, pipes are not synchronized across database

instances. Each database instance has its own pipe, private to the memory of the database instance.

A single message can be cached in a singleton pipe, which can be comprised of multiple fields up to a total message size of 32,767 bytes. DBMS_PIPE supports the ability to pack multiple attributes in a message using the PACK_MESSAGE procedure.

Singleton pipes can be public or private and implicit or explicit. Using a private singleton pipe, the message can only be received by sessions with the same user as the creator of the pipe. The message in a public singleton pipe can be received by any database session with EXECUTE privilege on the DBMS_PIPE package.

Singleton pipes cache the message in the pipe until it is invalidated or purged. Explicit invalidation by purging the pipe is accomplished by using the PURGE procedure or by overwriting the message using the SEND_MESSAGE function. Automatic invalidation happens once the shelflife time has elapsed, which is specified as part of the CREATE_PIPE function and the SEND_MESSAGE function.

A user-defined cache function can be used to automatically populate the message in a singleton pipe and can be specified when reading a message using the RECEIVE_MESSAGE function. If the message is not in cache or the message shelflife time has elapsed, the singleton pipe automatically populates a new message in the pipe. The cache function simplifies the use of a pipe by avoiding condition logic on failure to receive a message from an empty pipe. It also ensures there is no cache miss when reading messages from a singleton pipe.

> **Note:**
>
> When using a cache function with RECEIVE_MESSAGE, the time spent executing the cache function is not included towards the overall timeout specified for receiving messages.

Singleton pipes do not get evicted from Oracle Database memory, ensuring maximum cache hits. An explicit singleton pipe exists in database memory until it is removed using the REMOVE_PIPE procedure or the database restarts. An implicit singleton pipe exists in database memory until there is one cached message in the pipe.

Persistent Pipes

You can optionally use Cloud Object stores for persistent message storage. The process for sending and receiving messages from a pipe is largely the same whether you use local or cloud storage, the difference being with cloud storage, a credential object and associated cloud URI must be provided. DBMS_PIPE uses the DBMS_CLOUD package to integrate with object storage, meaning DBMS_PIPE supports all object stores and credential types that are allowed with DBMS_CLOUD.

Storing pipe messages in object storage allows two or more databases in the same or different regions to communicate using DBMS_PIPE messages and removes the max number of messages that can be stored in a pipe. This means that a database instance open on one Real Application Cluster (RAC) instance can reliably use DBMS_PIPE for communicating between processes running on different cluster instances.

A locking mechanism is implemented to allow only one sending or receiving operation to be working in the cloud object store at a time. A lock file is created when a process begins using the cloud store and then is deleted once the operation is complete. Only when there is no lock

present is a process able to begin working in the cloud store. Processes will retry until the given max wait time is reached. If a lock has been left behind by an unknown process that is no longer operating, the lock will be deleted and remade by the current process trying to access the cloud store after 15 minutes.

> **Note:**
>
> When accessing `DBMS_PIPE` across different databases using Cloud Object stores, Oracle recommends using the `CREATE_PIPE` function before sending or receiving a message to ensure that the pipe is created with the required access permissions of a private or public pipe.

> **See Also:**
>
> - *Using Oracle Autonomous Database Serverless* for supported URI formats
> - *Using Oracle Autonomous Database Serverless* for `DBMS_CLOUD` subprograms and REST APIs
> - *Using Oracle Autonomous Database Serverless* for information about supported credential types

DBMS_PIPE Exceptions

`DBMS_PIPE` package subprograms can return the errors listed in the following table.

Table 149-1 DBMS_PIPE Errors

Error	Description
`ORA-23321:`	Pipename may not be null. This can be returned by the `CREATE_PIPE` function, or any subprogram that takes a pipe name as a parameter.
`ORA-23322:`	Insufficient privilege to access pipe. This can be returned by any subprogram that references a private pipe in its parameter list.

DBMS_PIPE Examples

These examples show use of DBMS_PIPE in debugging PL/SQL, debugging Pro*C, executing system commands, and an external service interface.

Example 1: Debugging - PL/SQL

This example shows the procedure that a PL/SQL program can call to place debugging information in a pipe.

```
CREATE OR REPLACE PROCEDURE debug (msg VARCHAR2) AS
    status  NUMBER;
BEGIN
  DBMS_PIPE.PACK_MESSAGE(LENGTH(msg));
  DBMS_PIPE.PACK_MESSAGE(msg);
  status := DBMS_PIPE.SEND_MESSAGE('plsql_debug');
```

```
  IF status != 0 THEN
    raise_application_error(-20099, 'Debug error');
  END IF;
END debug;
```

Example 2: Debugging - Pro*C

The following Pro*C code receives messages from the `PLSQL_DEBUG` pipe in the previous example, and displays the messages. If the Pro*C session is run in a separate window, then it can be used to display any messages that are sent to the debug procedure from a PL/SQL program executing in a separate session.

```
#include <stdio.h>
#include <string.h>

EXEC SQL BEGIN DECLARE SECTION;
   VARCHAR username[20];
   int     status;
   int     msg_length;
   char    retval[2000];
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA;

void sql_error();

main()
{

-- Prepare username:
   strcpy(username.arr, "HR/<password>");
   username.len = strlen(username.arr);

   EXEC SQL WHENEVER SQLERROR DO sql_error();
   EXEC SQL CONNECT :username;

   printf("connected\n");

-- Start an endless loop to look for and print messages on the pipe:
   FOR (;;)
   {
      EXEC SQL EXECUTE
         DECLARE
            len INTEGER;
            typ INTEGER;
            sta INTEGER;
            chr VARCHAR2(2000);
         BEGIN
            chr := '';
            sta := dbms_pipe.receive_message('plsql_debug');
            IF sta = 0 THEN
               DBMS_PIPE.UNPACK_MESSAGE(len);
               DBMS_PIPE.UNPACK_MESSAGE(chr);
            END IF;
            :status := sta;
            :retval := chr;
            IF len IS NOT NULL THEN
               :msg_length := len;
            ELSE
               :msg_length := 2000;
            END IF;
         END;
```

```
      END-EXEC;
      IF (status == 0)
         printf("\n%.*s\n", msg_length, retval);
      ELSE
         printf("abnormal status, value is %d\n", status);
   }
}

void sql_error()
{
   char msg[1024];
   int rlen, len;
   len = sizeof(msg);
   sqlglm(msg, &len, &rlen);
   printf("ORACLE ERROR\n");
   printf("%.*s\n", rlen, msg);
   exit(1);
}
```

Example 3: Execute System Commands

This example shows PL/SQL and Pro*C code let a PL/SQL stored procedure (or anonymous block) call PL/SQL procedures to send commands over a pipe to a Pro*C program that is listening for them.

The Pro*C program sleeps and waits for a message to arrive on the named pipe. When a message arrives, the Pro*C program processes it, carrying out the required action, such as executing a UNIX command through the *system*() call or executing a SQL command using embedded SQL.

`DAEMON.SQL` is the source code for the PL/SQL package. This package contains procedures that use the `DBMS_PIPE` package to send and receive message to and from the Pro*C daemon. Note that full handshaking is used. The daemon always sends a message back to the package (except in the case of the `STOP` command). This is valuable, because it allows the PL/SQL procedures to be sure that the Pro*C daemon is running.

You can call the DAEMON packaged procedures from an anonymous PL/SQL block using SQL*Plus or Enterprise Manager. For example:

```
SQLPLUS> variable rv number
SQLPLUS> execute :rv := DAEMON.EXECUTE_SYSTEM('ls -la');
```

On a UNIX system, this causes the Pro*C daemon to execute the command *system*("*ls -la*").

Remember that the daemon needs to be running first. You might want to run it in the background, or in another window beside the SQL*Plus or Enterprise Manager session from which you call it.

The `DAEMON.SQL` also uses the `DBMS_OUTPUT` package to display the results. For this example to work, you must have execute privileges on this package.

DAEMON.SQL Example. This is the code for the PL/SQL `DAEMON` package:

```
CREATE OR REPLACE PACKAGE daemon AS
  FUNCTION execute_sql(command VARCHAR2,
                       timeout NUMBER DEFAULT 10)
    RETURN NUMBER;

  FUNCTION execute_system(command VARCHAR2,
                          timeout NUMBER DEFAULT 10)
    RETURN NUMBER;
```

```
  PROCEDURE stop(timeout NUMBER DEFAULT 10);
END daemon;
/
CREATE OR REPLACE PACKAGE BODY daemon AS

  FUNCTION execute_system(command VARCHAR2,
                          timeout NUMBER DEFAULT 10)
  RETURN NUMBER IS

    status       NUMBER;
    result       VARCHAR2(20);
    command_code NUMBER;
    pipe_name    VARCHAR2(30);
  BEGIN
    pipe_name := DBMS_PIPE.UNIQUE_SESSION_NAME;

    DBMS_PIPE.PACK_MESSAGE('SYSTEM');
    DBMS_PIPE.PACK_MESSAGE(pipe_name);
    DBMS_PIPE.PACK_MESSAGE(command);
    status := DBMS_PIPE.SEND_MESSAGE('daemon', timeout);
    IF status <> 0 THEN
      RAISE_APPLICATION_ERROR(-20010,
        'Execute_system: Error while sending.  Status = ' ||
         status);
    END IF;

    status := DBMS_PIPE.RECEIVE_MESSAGE(pipe_name, timeout);
    IF status <> 0 THEN
      RAISE_APPLICATION_ERROR(-20011,
        'Execute_system: Error while receiving.
         Status = ' || status);
    END IF;

    DBMS_PIPE.UNPACK_MESSAGE(result);
    IF result <> 'done' THEN
      RAISE_APPLICATION_ERROR(-20012,
        'Execute_system: Done not received.');
    END IF;

    DBMS_PIPE.UNPACK_MESSAGE(command_code);
    DBMS_OUTPUT.PUT_LINE('System command executed.  result = ' ||
                         command_code);
    RETURN command_code;
  END execute_system;

  FUNCTION execute_sql(command VARCHAR2,
                       timeout NUMBER DEFAULT 10)
  RETURN NUMBER IS

    status       NUMBER;
    result       VARCHAR2(20);
    command_code NUMBER;
    pipe_name    VARCHAR2(30);

  BEGIN
    pipe_name := DBMS_PIPE.UNIQUE_SESSION_NAME;

    DBMS_PIPE.PACK_MESSAGE('SQL');
    DBMS_PIPE.PACK_MESSAGE(pipe_name);
    DBMS_PIPE.PACK_MESSAGE(command);
    status := DBMS_PIPE.SEND_MESSAGE('daemon', timeout);
    IF status <> 0 THEN
```

```
      RAISE_APPLICATION_ERROR(-20020,
        'Execute_sql: Error while sending.  Status = ' || status);
    END IF;

    status := DBMS_PIPE.RECEIVE_MESSAGE(pipe_name, timeout);

    IF status <> 0 THEN
      RAISE_APPLICATION_ERROR(-20021,
        'execute_sql: Error while receiving.
         Status = ' || status);
    END IF;

    DBMS_PIPE.UNPACK_MESSAGE(result);
    IF result <> 'done' THEN
      RAISE_APPLICATION_ERROR(-20022,
        'execute_sql: done not received.');
    END IF;

    DBMS_PIPE.UNPACK_MESSAGE(command_code);
    DBMS_OUTPUT.PUT_LINE
        ('SQL command executed.  sqlcode = ' || command_code);
    RETURN command_code;
  END execute_sql;

  PROCEDURE stop(timeout NUMBER DEFAULT 10) IS
    status NUMBER;
  BEGIN
    DBMS_PIPE.PACK_MESSAGE('STOP');
    status := DBMS_PIPE.SEND_MESSAGE('daemon', timeout);
    IF status <> 0 THEN
      RAISE_APPLICATION_ERROR(-20030,
        'stop: error while sending.  status = ' || status);
    END IF;
  END stop;
END daemon;
```

daemon.pc Example. This is the code for the Pro*C daemon. You must precompile this using the Pro*C Precompiler, Version 1.5.x or later. You must also specify the `USERID` and `SQLCHECK` options, as the example contains embedded PL/SQL code.

> **Note:**
>
> To use a `VARCHAR` output host variable in a PL/SQL block, you must initialize the length component before entering the block.

```
proc iname=daemon userid=HR/<password> sqlcheck=semantics
```

Then C-compile and link in the normal way.

```
#include <stdio.h>
#include <string.h>

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
  char *uid = "HR/<password>";
  int status;
  VARCHAR command[20];
```

```
  VARCHAR value[2000];
  VARCHAR return_name[30];
EXEC SQL END DECLARE SECTION;

void
connect_error()
{
  char msg_buffer[512];
  int msg_length;
  int buffer_size = 512;

  EXEC SQL WHENEVER SQLERROR CONTINUE;
  sqlglm(msg_buffer, &buffer_size, &msg_length);
  printf("Daemon error while connecting:\n");
  printf("%.*s\n", msg_length, msg_buffer);
  printf("Daemon quitting.\n");
  exit(1);
}

void
sql_error()
{
  char msg_buffer[512];
  int msg_length;
  int buffer_size = 512;

  EXEC SQL WHENEVER SQLERROR CONTINUE;
  sqlglm(msg_buffer, &buffer_size, &msg_length);
  printf("Daemon error while executing:\n");
  printf("%.*s\n", msg_length, msg_buffer);
  printf("Daemon continuing.\n");
}
main()
{
command.len = 20; /*initialize length components*/
value.len = 2000;
return_name.len  = 30;
  EXEC SQL WHENEVER SQLERROR DO connect_error();
  EXEC SQL CONNECT :uid;
  printf("Daemon connected.\n");

  EXEC SQL WHENEVER SQLERROR DO sql_error();
  printf("Daemon waiting...\n");
  while (1) {
    EXEC SQL EXECUTE
      BEGIN
        :status := DBMS_PIPE.RECEIVE_MESSAGE('daemon');
        IF :status = 0 THEN
          DBMS_PIPE.UNPACK_MESSAGE(:command);
        END IF;
      END;
    END-EXEC;
    IF (status == 0)
    {
      command.arr[command.len] = '\0';
      IF (!strcmp((char *) command.arr, "STOP"))
      {
        printf("Daemon exiting.\n");
        break;
      }

      ELSE IF (!strcmp((char *) command.arr, "SYSTEM"))
```

```
  {
    EXEC SQL EXECUTE
      BEGIN
        DBMS_PIPE.UNPACK_MESSAGE(:return_name);
        DBMS_PIPE.UNPACK_MESSAGE(:value);
      END;
    END-EXEC;
    value.arr[value.len] = '\0';
    printf("Will execute system command '%s'\n", value.arr);

    status = system(value.arr);
    EXEC SQL EXECUTE
      BEGIN
        DBMS_PIPE.PACK_MESSAGE('done');
        DBMS_PIPE.PACK_MESSAGE(:status);
        :status := DBMS_PIPE.SEND_MESSAGE(:return_name);
      END;
    END-EXEC;

    IF (status)
    {
      printf
       ("Daemon error while responding to system command.");
      printf("  status: %d\n", status);
    }
  }
  ELSE IF (!strcmp((char *) command.arr, "SQL")) {
    EXEC SQL EXECUTE
      BEGIN
        DBMS_PIPE.UNPACK_MESSAGE(:return_name);
        DBMS_PIPE.UNPACK_MESSAGE(:value);
      END;
    END-EXEC;
    value.arr[value.len] = '\0';
    printf("Will execute sql command '%s'\n", value.arr);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL EXECUTE IMMEDIATE :value;
    status = sqlca.sqlcode;

    EXEC SQL WHENEVER SQLERROR DO sql_error();
    EXEC SQL EXECUTE
      BEGIN
        DBMS_PIPE.PACK_MESSAGE('done');
        DBMS_PIPE.PACK_MESSAGE(:status);
        :status := DBMS_PIPE.SEND_MESSAGE(:return_name);
      END;
    END-EXEC;

    IF (status)
    {
      printf("Daemon error while responding to sql command.");
      printf("  status: %d\n", status);
    }
  }
  ELSE
  {
    printf
      ("Daemon error: invalid command '%s' received.\n",
        command.arr);
  }
}
```

```
  ELSE
  {
    printf("Daemon error while waiting for signal.");
    printf("  status = %d\n", status);
  }
}
EXEC SQL COMMIT WORK RELEASE;
exit(0);
```

Example 4: External Service Interface

Put the user-written 3GL code into an OCI or Precompiler program. The program connects to the database and executes PL/SQL code to read its request from the pipe, computes the result, and then executes PL/SQL code to send the result on a pipe back to the requestor.

Below is an example of a stock service request. The recommended sequence for the arguments to pass on the pipe for all service requests is:

```
  protocol_version      VARCHAR2        - '1', 10 bytes or less
  returnpipe            VARCHAR2        - 30 bytes or less
  service               VARCHAR2        - 30 bytes or less
  arg1                  VARCHAR2/NUMBER/DATE
     ...
  argn                  VARCHAR2/NUMBER/DATE
```

The recommended format for returning the result is:

```
  success               VARCHAR2        - 'SUCCESS' if OK,
                                          otherwise error message
  arg1                  VARCHAR2/NUMBER/DATE
     ...
  argn                  VARCHAR2/NUMBER/DATE
```

The "stock price request server" would do, using OCI or PRO* (in pseudo-code):

```
<loop forever>
  BEGIN dbms_stock_server.get_request(:stocksymbol); END;
  <figure out price based on stocksymbol (probably from some radio
        signal), set error if can't find such a stock>
  BEGIN dbms_stock_server.return_price(:error, :price); END;
```

A client would do:

```
BEGIN :price := stock_request('YOURCOMPANY'); end;
```

The stored procedure, `dbms_stock_server`, which is called by the preceding "stock price request server" is:

```
CREATE OR REPLACE PACKAGE dbms_stock_server IS
  PROCEDURE get_request(symbol OUT VARCHAR2);
  PROCEDURE return_price(errormsg IN VARCHAR2, price IN VARCHAR2);
END;

CREATE OR REPLACE PACKAGE BODY dbms_stock_server IS
  returnpipe    VARCHAR2(30);

  PROCEDURE returnerror(reason VARCHAR2) IS
    s INTEGER;
  BEGIN
    dbms_pipe.pack_message(reason);
    s := dbms_pipe.send_message(returnpipe);
    IF s <> 0 THEN
      raise_application_error(-20000, 'Error:' || to_char(s) ||
```

```
        ' sending on pipe');
    END IF;
  END;

  PROCEDURE get_request(symbol OUT VARCHAR2) IS
    protocol_version VARCHAR2(10);
    s                INTEGER;
    service          VARCHAR2(30);
  BEGIN
    s := dbms_pipe.receive_message('stock_service');
    IF s <> 0 THEN
      raise_application_error(-20000, 'Error:' || to_char(s) ||
        'reading pipe');
    END IF;
    dbms_pipe.unpack_message(protocol_version);
    IF protocol_version <> '1' THEN
      raise_application_error(-20000, 'Bad protocol: ' ||
        protocol_version);
    END IF;
    dbms_pipe.unpack_message(returnpipe);
    dbms_pipe.unpack_message(service);
    IF service != 'getprice' THEN
      returnerror('Service ' || service || ' not supported');
    END IF;
    dbms_pipe.unpack_message(symbol);
  END;

  PROCEDURE return_price(errormsg in VARCHAR2, price in VARCHAR2) IS
    s INTEGER;
  BEGIN
    IF errormsg is NULL THEN
      dbms_pipe.pack_message('SUCCESS');
      dbms_pipe.pack_message(price);
    ELSE
      dbms_pipe.pack_message(errormsg);
    END IF;
    s := dbms_pipe.send_message(returnpipe);
    IF s <> 0 THEN
      raise_application_error(-20000, 'Error:'||to_char(s)||
        ' sending on pipe');
    END IF;
  END;
END;
```

The procedure called by the client is:

```
CREATE OR REPLACE FUNCTION stock_request (symbol VARCHAR2)
    RETURN VARCHAR2 IS
  s        INTEGER;
  price    VARCHAR2(20);
  errormsg VARCHAR2(512);
BEGIN
  dbms_pipe.pack_message('1');  -- protocol version
  dbms_pipe.pack_message(dbms_pipe.unique_session_name); -- return pipe
  dbms_pipe.pack_message('getprice');
  dbms_pipe.pack_message(symbol);
  s := dbms_pipe.send_message('stock_service');
  IF s <> 0 THEN
    raise_application_error(-20000, 'Error:'||to_char(s)||
      ' sending on pipe');
  END IF;
```

```
  s := dbms_pipe.receive_message(dbms_pipe.unique_session_name);
  IF s <> 0 THEN
    raise_application_error(-20000, 'Error:'||to_char(s)||
      ' receiving on pipe');
  END IF;
  dbms_pipe.unpack_message(errormsg);
  IF errormsg <> 'SUCCESS' THEN
    raise_application_error(-20000, errormsg);
  END IF;
  dbms_pipe.unpack_message(price);
  RETURN price;
END;
```

You would typically only `GRANT EXECUTE` on `DBMS_STOCK_SERVICE` to the stock service application server, and would only `GRANT EXECUTE` on `stock_request` to those users allowed to use the service.

> **See Also:**
>
> DBMS_ALERT

Summary of DBMS_PIPE Subprograms

This table lists the `DBMS_PIPE` subprograms and briefly describes them.

Table 149-2 DBMS_PIPE Package Subprograms

Subprogram	Description
CREATE_PIPE Function	Creates a pipe (necessary for private pipes)
GET_CREDENTIAL_NAME Function	Returns the globally set `credential_name` variable
GET_LOCATION_URI Function	Returns the globally set `location_uri` variable
NEXT_ITEM_TYPE Function	Returns datatype of next item in buffer
PACK_MESSAGE Procedures	Builds message in local buffer
PURGE Procedure	Purges contents of named pipe
RECEIVE_MESSAGE Function	Copies message from named pipe into local buffer
REMOVE_PIPE Function	Removes the named pipe
RESET_BUFFER Procedure	Purges contents of local buffer
SEND_MESSAGE Function	Sends message on named pipe: This implicitly creates a public pipe if the named pipe does not exist
SET_CREDENTIAL_NAME Procedure	Sets the global `credential_name` variable
SET_LOCATION_URI Procedure	Sets the global `location_uri` variable
UNIQUE_SESSION_NAME Function	Returns unique session name
UNPACK_MESSAGE Procedures	Accesses next item in buffer

CREATE_PIPE Function

This function explicitly creates a public or private pipe. If the `private` flag is `TRUE`, then the pipe creator is assigned as the owner of the private pipe.

Explicitly-created pipes can only be removed by calling `REMOVE_PIPE`, or by shutting down the instance.

Syntax

```
DBMS_PIPE.CREATE_PIPE (
   pipename     IN VARCHAR2,
   maxpipesize  IN INTEGER DEFAULT 65536,
   private      IN BOOLEAN DEFAULT TRUE,
   singleton    IN BOOLEAN DEFAULT FALSE,
   shelflife    IN INTEGER DEFAULT 0)
RETURN INTEGER;
```

Pragmas

```
pragma restrict_references(create_pipe,WNDS,RNDS);
```

Parameters

Table 149-3 CREATE_PIPE Function Parameters

Parameter	Description
`pipename`	Name of the pipe you are creating. You must use this name when you call `SEND_MESSAGE` and `RECEIVE_MESSAGE`. This name must be unique across the instance. Caution: Do not use pipe names beginning with `ORA$`. These are reserved for use by procedures provided by Oracle. Pipename should not be longer than 128 bytes, and is case insensitive. At this time, the name cannot contain Globalization Support characters.
`maxpipesize`	The maximum size allowed for the pipe, in bytes. The total size of all of the messages on the pipe cannot exceed this amount. The message is blocked if it exceeds this maximum. The default `maxpipesize` is 65536 bytes. The `maxpipesize` for a pipe becomes a part of the characteristics of the pipe and persists for the life of the pipe. Callers of `SEND_MESSAGE` with larger values cause the `maxpipesize` to be increased. Callers with a smaller value use the existing, larger value.
`private`	Uses the default, `TRUE`, to create a private pipe. Public pipes can be implicitly created when you call `SEND_MESSAGE`.
`singleton`	Use the value `TRUE` to indicate that the pipe should be created as a singleton pipe. Singleton pipes cannot be persistent pipes. The default value of `singleton` is `FALSE`.

Table 149-3 (Cont.) CREATE_PIPE Function Parameters

Parameter	Description
shelflife	Only applicable to singleton pipes, this parameter is optionally used to set the expiration time in seconds of a message cached in a singleton pipe. Once the shelflife time is exceeded, the message is no longer accessible from the pipe. The shelflife can be used for implicit invalidation of the message in a singleton pipe. The default value of 0 indicates that the message will not expire. The shelflife of a message in a singleton pipe can also be specified when sending a message. See SEND_MESSAGE Function.

Return Values

Table 149-4 CREATE_PIPE Function Return Values

Return	Description
0	Successful. If the pipe already exists and the user attempting to create it is authorized to use it, then Oracle returns 0, indicating success, and any data already in the pipe remains. If a user connected as SYSDBA/SYSOPER re-creates a pipe, then Oracle returns status 0, but the ownership of the pipe remains unchanged.
6	Failed to convert existing pipe to a singleton pipe. An implicit pipe with more than one existing message cannot be converted to a singleton pipe. For an explicit pipe that is not a singleton pipe, SEND_MESSAGE cannot send a message with the singleton parameter set to TRUE.
7	A non-zero value was given for shelflife and the pipe is not a singleton pipe.
ORA-23322	Failure due to naming conflict. If a pipe with the same name exists and was created by a different user, then Oracle signals error ORA-23322, indicating the naming conflict.

Exceptions

Table 149-5 CREATE_PIPE Function Exception

Exception	Description
Null pipe name	Permission error: Pipe with the same name already exists, and you are not allowed to use it.

GET_CREDENTIAL_NAME Function

This function retrieves the global `credential_name` variable to be used as the default credential with the cloud. The `GET_CREDENTIAL_NAME` function is only applicable to persistent pipes with messages stored in Cloud Object Storage.

Syntax

```
DBMS_PIPE.GET_CREDENTIAL_NAME ()
RETURN VARCHAR2;
```

Return Values

This function returns the globally set `credential_name` variable. The variable is set to `NULL` by default.

GET_LOCATION_URI Function

This function retrieves the global `location_uri` variable to be used as the default location URI with the cloud. The `GET_LOCATION_URI` function is only applicable to persistent pipes with messages stored in Cloud Object Storage.

Syntax

```
DBMS_PIPE.GET_LOCATION_URI ()
RETURN VARCHAR2;
```

Return Values

This function returns the globally set `location_uri` variable. The variable is set to `NULL` by default.

NEXT_ITEM_TYPE Function

This function determines the datatype of the next item in the local message buffer.

After you have called `RECEIVE_MESSAGE` to place pipe information in a local buffer, call `NEXT_ITEM_TYPE`.

Syntax

```
DBMS_PIPE.NEXT_ITEM_TYPE
  RETURN INTEGER;
```

Pragmas

```
pragma restrict_references(next_item_type,WNDS,RNDS);
```

Return Values

Table 149-6 NEXT_ITEM_TYPE Function Return Values

Return	Description
0	No more items
6	NUMBER
9	VARCHAR2
11	ROWID
12	DATE
23	RAW

PACK_MESSAGE Procedures

This procedure builds your message in the local message buffer.

To send a message, first make one or more calls to `PACK_MESSAGE`. Then, call `SEND_MESSAGE` to send the message in the local buffer on the named pipe.

The procedure is overloaded to accept items of type `VARCHAR2`, `NCHAR`, `NUMBER`, `DATE`., `RAW` and `ROWID` items. In addition to the data bytes, each item in the buffer requires one byte to indicate its type, and two bytes to store its length. One additional byte is needed to terminate the message.The overhead for all types other than `VARCHAR` is 4 bytes.

Syntax

```
DBMS_PIPE.PACK_MESSAGE (
   item  IN  VARCHAR2);

DBMS_PIPE.PACK_MESSAGE (
   item  IN  NCHAR);

DBMS_PIPE.PACK_MESSAGE (
   item  IN  NUMBER);

DBMS_PIPE.PACK_MESSAGE (
   item  IN  DATE);

DBMS_PIPE.PACK_MESSAGE_RAW (
   item  IN  RAW);

DBMS_PIPE.PACK_MESSAGE_ROWID (
   item  IN  ROWID);
```

Pragmas

```
pragma restrict_references(pack_message,WNDS,RNDS);
pragma restrict_references(pack_message_raw,WNDS,RNDS);
pragma restrict_references(pack_message_rowid,WNDS,RNDS);
```

Parameters

Table 149-7 PACK_MESSAGE Procedure Parameters

Parameter	Description
item	Item to pack into the local message buffer.

Usage Notes

In Oracle database version 8.x, the char-set-id (2 bytes) and the char-set-form (1 byte) are stored with each data item. Therefore, the overhead when using Oracle database version 8.x is 7 bytes.

When you call SEND_MESSAGE to send this message, you must indicate the name of the pipe on which you want to send the message. If this pipe already exists, then you must have sufficient privileges to access this pipe. If the pipe does not already exist, then it is created automatically.

Exceptions

ORA-06558 is raised if the message buffer overflows (currently 4096 bytes). Each item in the buffer takes one byte for the type, two bytes for the length, plus the actual data. There is also one byte needed to terminate the message.

PURGE Procedure

This procedure empties the contents of the named pipe.

An empty implicitly-created pipe is aged out of the shared global area according to the least-recently-used algorithm. Thus, calling PURGE lets you free the memory associated with an implicitly-created pipe.

Syntax

```
DBMS_PIPE.PURGE (
   pipename  IN  VARCHAR2);
```

Pragmas

```
pragma restrict_references(purge,WNDS,RNDS);
```

Parameters

Table 149-8 PURGE Procedure Parameters

Parameter	Description
pipename	Name of pipe from which to remove all messages. The local buffer may be overwritten with messages as they are discarded. Pipename should not be longer than 128 bytes, and is case-insensitive.

Usage Notes

Because `PURGE` calls `RECEIVE_MESSAGE`, the local buffer might be overwritten with messages as they are purged from the pipe. Also, you can receive an `ORA-23322` (insufficient privileges) error if you attempt to purge a pipe with which you have insufficient access rights.

Exceptions

Permission error if pipe belongs to another user.

RECEIVE_MESSAGE Function

This function copies the message into the local message buffer.

Syntax

```
DBMS_PIPE.RECEIVE_MESSAGE (
   pipename     IN VARCHAR2,
   timeout      IN INTEGER      DEFAULT maxwait,
   cache_func   IN VARCHAR2     DEFAULT NULL)
RETURN INTEGER;


DBMS_PIPE.RECEIVE_MESSAGE (
   pipename           IN VARCHAR2,
   timeout            IN INTEGER      DEFAULT maxwait,
   credential_name    IN VARCHAR2     DEFAULT NULL,
   location_uri       IN VARCHAR2     DEFAULT NULL)
RETURN INTEGER;
```

Pragmas

```
pragma restrict_references(receive_message,WNDS,RNDS);
```

Parameters

Table 149-9 RECEIVE_MESSAGE Function Parameters

Parameter	Description
`pipename`	Name of the pipe on which you want to receive a message. Names beginning with `ORA$` are reserved for use by Oracle
`timeout`	Time to wait for a message, in seconds. The default value is the constant `MAXWAIT`, which is defined as 86400000 (1000 days). A timeout of 0 lets you read without blocking.

Table 149-9 (Cont.) RECEIVE_MESSAGE Function Parameters

Parameter	Description
cache_func	Only applicable to singleton pipes, cache_func is the cache function name used to automatically cache a message in a singleton pipe and can be created as either a PL/SQL function or an embedded function in a PL/SQL package. If specified, the cache function is invoked as the current session user invoking the RECEIVE_MESSAGE function so the current user must have privilege on the function. The name of the function must be fully qualified with the owner schema: • OWNER.FUNCTION_NAME • OWNER.PACKAGE.FUNCTION_NAME Note that the time spent executing the cache function is not included towards the overall timeout specified for receiving messages.
credential_name	The credential name for the cloud store used to store messages. This parameter is only applicable to persistent pipes. The default value is NULL. A passed parameter takes precedence over the package argument's value. Credentials require EXECUTE and READ/WRITE privileges.
location_uri	The location URL for the cloud store being used to store messages. This parameter is only applicable to persistent pipes. The location_uri parameter is a global variable that has a value of NULL by default. A passed parameter takes precedence over the global variable's value. If a credential name is specified, a location_uri must also be provided.

Return Values

Table 149-10 RECEIVE_MESSAGE Function Return Values

Return	Description
0	Success
1	Timed out. If the pipe was implicitly-created and is empty, then it is removed.
2	Record in the pipe is too large for the buffer. (This should not happen.)
3	An interrupt occurred.
8	The cache function was specified on a non-singleton pipe.
ORA-23322	User has insufficient privileges to read from the pipe.

Usage Notes

To receive a message from a pipe, first call RECEIVE_MESSAGE. When you receive a message, it is removed from the pipe; hence, a message can only be received once (unless using a singleton pipe). For implicitly-created pipes, the pipe is removed after the last record is removed from the pipe. An implicit singleton pipe exists in database memory until there is one cached message in the pipe.

If the pipe that you specify when you call `RECEIVE_MESSAGE` does not already exist, then Oracle implicitly creates the pipe and waits to receive the message. If the message does not arrive within a designated timeout interval, then the call returns and the pipe is removed.

After receiving the message, you must make one or more calls to `UNPACK_MESSAGE` to access the individual items in the message. The `UNPACK_MESSAGE` procedure is overloaded to unpack items of type `DATE`, `NUMBER`, `VARCHAR2`, and there are two additional procedures to unpack `RAW` and `ROWID` items. If you do not know the type of data that you are attempting to unpack, then call `NEXT_ITEM_TYPE` to determine the type of the next item in the buffer.

> **Note:**
>
> If performing cross-database messaging, the `CREATE_PIPE` function must be called before the first time you attempt to receive a message in the new database. The name of this pipe and its properties must be the same as the pipe the message was sent on in the other database.

Persistent messages are guaranteed to either be written or read by exactly one process. This prevents message content inconsistency due to concurrent writes and reads. Using a persistent messaging pipe, `DBMS_PIPE` allows only one operation, sending a message or receiving a message to be active at a given time. However, if an operation is not possible due to an ongoing operation, the process retries periodically until the `timeout` value is reached.

If you use Oracle Cloud Infrastructure Object Storage to store messages, you can use Oracle Cloud Infrastructure Native URIs or Swift URIs. However, the location URI and the credential must match in type as follows:

- If you use a native URI format to access Oracle Cloud Infrastructure Object Storage, you must use Native Oracle Cloud Infrastructure Signing Keys authentication in the credential object.
- If you use Swift URI format to access Oracle Cloud Infrastructure Object Storage, you must use an auth token authentication in the credential object.

Cache Function Parameter

Singleton pipes support the cache function to automatically cache a message in the pipe in case of the following two scenarios:

- The singleton pipe is empty
- The message in the singleton pipe is invalid because the `shelflife` time has elapsed

The cache function simplifies the use of singleton pipes by avoiding condition logic on failure to receive a message from an empty pipe and ensures there is no cache miss.

The name of the function should be fully qualified with the owner schema:

- `OWNER.FUNCTION_NAME`
- `OWNER.PACKAGE.FUNCTION_NAME`

To use a cache function, the current session user that invokes `DBMS_PIPE.RECEIVE_MESSAGE` must have required privileges to execute the cache function.

Cache Function Syntax

```
CREATE OR REPLACE FUNCTION cache_function_name(
   pipename IN VARCHAR2
) RETURN INTEGER;
```

Cache Function Parameters

Parameter	Data Type	Description
pipename	VARCHAR2	Name of the singleton pipe

Cache Function Return Values

Return	Description
0	Success
Non-zero	Failure value returned from DBMS_PIPE.RECEIVE_MESSAGE

Define a cache function to provide encapsulation and abstraction of complexity from the reader sessions of a singleton pipe. The typical operations within a cache function would be:

- Create a singleton pipe for an explicit pipe using DBMS_PIPE.CREATE_PIPE
- Create the message to cache in the singleton pipe
- Send the message to the singleton pipe, optionally specifying a shelflife for the implicit message

Exceptions

Table 149-11 RECEIVE_MESSAGE Function Exceptions

Exception	Description
Null pipe name	Permission error. Insufficient privilege to remove the record from the pipe. The pipe is owned by someone else.

Example

```
DECLARE
   l_status INTEGER;
BEGIN
   l_status := DBMS_PIPE.RECEIVE_MESSAGE(pipename   => 'MY_PIPE1',
                                         timeout    => 1,
                                         cache_func =>
'MY_USER.MY_CACHE_FUNC');
END;
/
```

RESET_BUFFER Procedure

This procedure resets the PACK_MESSAGE and UNPACK_MESSAGE positioning indicators to 0.

Because all pipes share a single buffer, you may find it useful to reset the buffer before using a new pipe. This ensures that the first time you attempt to send a message to your pipe, you do not inadvertently send an expired message remaining in the buffer.

Syntax

```
DBMS_PIPE.RESET_BUFFER;
```

Pragmas

```
pragma restrict_references(reset_buffer,WNDS,RNDS);
```

REMOVE_PIPE Function

This function removes explicitly-created pipes.

Pipes created implicitly by SEND_MESSAGE are automatically removed when empty. However, pipes created explicitly by CREATE_PIPE are removed only by calling REMOVE_PIPE, or by shutting down the instance. All unconsumed records in the pipe are removed before the pipe is deleted.

This is similar to calling PURGE on an implicitly-created pipe.

Syntax

```
DBMS_PIPE.REMOVE_PIPE (
   pipename  IN  VARCHAR2)
RETURN INTEGER;
```

Pragmas

```
pragma restrict_references(remove_pipe,WNDS,RNDS);
```

Parameters

Table 149-12 REMOVE_PIPE Function Parameters

Parameter	Description
pipename	Name of pipe that you want to remove.

Return Values

Table 149-13 REMOVE_PIPE Function Return Values

Return	Description
0	Success If the pipe does not exist, or if the pipe already exists and the user attempting to remove it is authorized to do so, then Oracle returns 0, indicating success, and any data remaining in the pipe is removed.

Table 149-13 (Cont.) REMOVE_PIPE Function Return Values

Return	Description
ORA-23322	Insufficient privileges. If the pipe exists, but the user is not authorized to access the pipe, then Oracle signals error ORA-23322, indicating insufficient privileges.

Exceptions

Table 149-14 REMOVE_PIPE Function Exception

Exception	Description
Null pipe name	Permission error: Insufficient privilege to remove pipe. The pipe was created and is owned by someone else.

SEND_MESSAGE Function

This function sends a message on the named pipe.

The message is contained in the local message buffer, which was filled with calls to PACK_MESSAGE. You can create a pipe explicitly using CREATE_PIPE, otherwise, it is created implicitly.

To create an implicit singleton pipe, set the singleton parameter to TRUE. The following arguments are applicable to singleton pipes:

- singleton: Indicates that the pipe should be created as a singleton pipe (default: FALSE).
- shelflife: Optionally specify a shelflife expiration of a cached message in the singleton pipe. It can be used for implicit invalidation of the message in a singleton pipe. This argument is applicable for implicit as well as explicit singleton pipes. A shelflife value specified in the SEND_MESSAGE Function overwrites the shelflife specified for the explicit singleton pipe in the CREATE_PIPE Function and will be the default for any new messages cached in the singleton pipe.

Syntax

```
DBMS_PIPE.SEND_MESSAGE (
    pipename     IN VARCHAR2,
    timeout      IN INTEGER DEFAULT MAXWAIT,
    maxpipesize  IN INTEGER DEFAULT 65536,
    singleton    IN BOOLEAN DEFAULT FALSE,
    shelflife    IN INTEGER DEFAULT 0)
  RETURN INTEGER;


DBMS_PIPE.SEND_MESSAGE (
    pipename         IN VARCHAR2,
    timeout          IN INTEGER DEFAULT MAXWAIT,
    credential_name  IN VARCHAR2 DEFAULT NULL,
    location_uri     IN VARCHAR2 DEFAULT NULL)
  RETURN INTEGER;
```

Pragmas

```
pragma restrict_references(send_message,WNDS,RNDS);
```

Parameters

Table 149-15 SEND_MESSAGE Function Parameters

Parameter	Description
pipename	Name of the pipe on which you want to place the message. If you are using an explicit pipe, then this is the name that you specified when you called CREATE_PIPE. Caution: Do not use pipe names beginning with 'ORA$'. These names are reserved for use by procedures provided by Oracle. Pipename should not be longer than 128 bytes, and is case-insensitive. At this time, the name cannot contain Globalization Support characters.
timeout	Time to wait while attempting to place a message on a pipe, in seconds. The default value is the constant MAXWAIT, which is defined as 86400000 (1000 days).
maxpipesize	Maximum size allowed for the pipe, in bytes. The total size of all the messages on the pipe cannot exceed this amount. The message is blocked if it exceeds this maximum. The default is 65536 bytes. The maxpipesize for a pipe becomes a part of the characteristics of the pipe and persists for the life of the pipe. Callers of SEND_MESSAGE with larger values cause the maxpipesize to be increased. Callers with a smaller value simply use the existing, larger value. Specifying maxpipesize as part of the SEND_MESSAGE procedure eliminates the need for a separate call to open the pipe. If you created the pipe explicitly, then you can use the optional maxpipesize parameter to override the creation pipe size specifications.
singleton	Specifying singleton as TRUE indicates that the implicit pipe should be created as a singleton pipe. This argument is not required if the pipe is explicitly created as a singleton pipe using the CREATE_PIPE function. The default value of singleton is FALSE.
shelflife	Only applicable to singleton pipes, this parameter is optionally used to set the expiration time in seconds of a message cached in an implicit or explicit singleton pipe. Once the shelflife time is exceeded, the message is no longer accessible from the pipe. The shelflife can be used for implicit invalidation of the message in a singleton pipe. The default value of 0 indicates that the message will not expire. Specifying shelflife as part of the SEND_MESSAGE procedure overwrites the shelflife value specified for an explicit singleton pipe using CREATE_PIPE and will be the default for any new messages cached in the singleton pipe.
credential_name	The credential name for the cloud store used to store messages. This parameter is only applicable to persistent pipes. The default value is NULL. A passed parameter takes precedence over the package argument's value. Credentials require the EXECUTE and READ/WRITE privileges.

Table 149-15 (Cont.) SEND_MESSAGE Function Parameters

Parameter	Description
location_uri	The location URL for the cloud store being used to store messages. This parameter is only applicable to persistent pipes. The location_uri parameter is a global variable that has a value of NULL by default. A passed parameter takes precedence over the global variable's value. If a credential name is specified, a location_uri must also be provided.

Return Values

Table 149-16 SEND_MESSAGE Function Return Values

Return	Description
0	Success. If the pipe already exists and the user attempting to create it is authorized to use it, then Oracle returns 0, indicating success, and any data already in the pipe remains. If a user connected as SYSDBS/SYSOPER re-creates a pipe, then Oracle returns status 0, but the ownership of the pipe remains unchanged.
1	Timed out. This procedure can timeout either because it cannot get a lock on the pipe, or because the pipe remains too full to be used. If the pipe was implicitly-created and is empty, then it is removed.
3	An interrupt occurred. If the pipe was implicitly created and is empty, then it is removed.
6	Failed to convert existing pipe to a singleton pipe. An implicit pipe with more than one existing message cannot be converted to a singleton pipe. For explicit pipe that is not a singleton pipe, SEND_MESSAGE cannot send a message with the singleton parameter set to TRUE.
7	A non-zero value was given for shelflife and the pipe is not a singleton pipe.
ORA-23322	Insufficient privileges. If a pipe with the same name exists and was created by a different user, then Oracle signals error ORA-23322, indicating the naming conflict.

Usage Notes

Persistent messages are guaranteed to either be written or read by exactly one process. This prevents message content inconsistency due to concurrent writes and reads. Using a persistent messaging pipe, DBMS_PIPE allows only one operation, sending a message or receiving a message to be active at a given time. However, if an operation is not possible due to an ongoing operation, the process retries periodically until the timeout value is reached.

If you use Oracle Cloud Infrastructure Object Storage to store messages, you can use Oracle Cloud Infrastructure Native URIs or Swift URIs. However, the location URI and the credential must match in type as follows:

- If you use a native URI format to access Oracle Cloud Infrastructure Object Storage, you must use Native Oracle Cloud Infrastructure Signing Keys authentication in the credential object.
- If you use Swift URI format to access Oracle Cloud Infrastructure Object Storage, you must use an auth token authentication in the credential object.

Exceptions

Table 149-17 SEND_MESSAGE Function Exception

Exception	Description
`Null pipe name`	Permission error. Insufficient privilege to write to the pipe. The pipe is private and owned by someone else.

SET_CREDENTIAL_NAME Procedure

This procedure sets the global `credential_name` variable to be used as the default credential with the cloud. The `SET_CREDENTIAL_NAME` procedure is only applicable to persistent pipes with messages stored in Cloud Object Storage.

Syntax

```
DBMS_PIPE.SET_CREDENTIAL_NAME (
   credential_name    IN VARCHAR2);
```

Parameters

Table 149-18 SET_CREDENTIAL_NAME Procedure Parameters

Parameter	Description
`credential_name`	The credential name for the cloud store used to store messages.

SET_LOCATION_URI Procedure

This procedure sets the global `set_location_uri` variable to be used as the default location URI with the cloud. The `SET_LOCATION_URI` procedure is only applicable to persistent pipes with messages stored in Cloud Object Storage.

Syntax

```
DBMS_PIPE.SET_LOCATION_URI (
   location_uri    IN VARCHAR2);
```

Parameters

Table 149-19 SET_LOCATION_URI Procedure

Parameter	Description
location_uri	The location URI for the cloud store used to store messages.

UNIQUE_SESSION_NAME Function

This function receives a name that is unique among all of the sessions that are currently connected to a database.

Multiple calls to this function from the same session always return the same value. You might find it useful to use this function to supply the PIPENAME parameter for your SEND_MESSAGE and RECEIVE_MESSAGE calls.

Syntax

```
DBMS_PIPE.UNIQUE_SESSION_NAME
  RETURN VARCHAR2;
```

Pragmas

```
pragma restrict_references(unique_session_name,WNDS,RNDS,WNPS);
```

Return Values

This function returns a unique name. The returned name can be up to 30 bytes.

UNPACK_MESSAGE Procedures

This procedure retrieves items from the buffer.

After you have called RECEIVE_MESSAGE to place pipe information in a local buffer, call UNPACK_MESSAGE.

> **Note:**
>
> The UNPACK_MESSAGE procedure is overloaded to return items of type VARCHAR2, NCHAR, NUMBER, or DATE. There are two additional procedures to unpack RAW and ROWID items.

Syntax

```
DBMS_PIPE.UNPACK_MESSAGE (
   item  OUT VARCHAR2);

DBMS_PIPE.UNPACK_MESSAGE (
   item  OUT NCHAR);

DBMS_PIPE.UNPACK_MESSAGE (
   item  OUT NUMBER);
```

```
DBMS_PIPE.UNPACK_MESSAGE (
   item  OUT DATE);

DBMS_PIPE.UNPACK_MESSAGE_RAW (
   item  OUT RAW);

DBMS_PIPE.UNPACK_MESSAGE_ROWID (
   item  OUT ROWID);
```

Pragmas

```
pragma restrict_references(unpack_message,WNDS,RNDS);
pragma restrict_references(unpack_message_raw,WNDS,RNDS);
pragma restrict_references(unpack_message_rowid,WNDS,RNDS);
```

Parameters

Table 149-20 UNPACK_MESSAGE Procedure Parameters

Parameter	Description
item	Argument to receive the next unpacked item from the local message buffer.

Exceptions

ORA-06556 or 06559 are generated if the buffer contains no more items, or if the item is not of the same type as that requested.