DBMS_PIPE

As promised on page 1041 of the book, here is an example of a small pipe server, which answers the frequently asked question, 'How can I run a host command from PL/SQL?' With the addition of Java to the database and C external procedures, we could easily implement a host command function with either technology. However, what if we do not have access to a C compiler or we don't have the Java component of the database available – what then? The following shows how we could very simply set up a small **pipe server** that can do host commands using nothing more than SQL*PLUS and the `csh` scripting language. It is fairly simple, consisting of only a few lines of `csh` and even fewer of PL/SQL. It does show much of the power of database pipes though, and should give you some ideas for other interesting implementations.

We will start with the 'server' script. This part of the application is responsible for creating the private pipe on which we will listen for requests. We will usually use a private pipe here to prevent other malicious sessions from reading or writing on our pipe. This server stub will be responsible for reading the request from the pipe, processing the request, and writing the answer (the output of the host command) back to the client. The pseudo-code for this server is:

```
If passed a pipe name (every time except startup)
then
    write answer back to client on the pipe
end if

wait for request to be written on pipe
when request received, create a csh script to run the command,
    always make the last line of this csh script rerun THIS
    script, passing the name of the response pipe

exec the tmp script we just generated
```

The `host.csh` script (it must be named `host.csh`) for this will be:

```
#!/bin/csh -f

#
```

```
# If the first argument is supplied, then this is the
# name of the pipe on which to write the answer back to the client.
# The answer they are waiting for is in the file
# $$.dat - $$ will be the pid (process id) of our csh
# script. We use EXEC to ensure the pid stays the same
# from call to call.

if ( "$1" != "" ) then

    #
    # If we get here, we will get the 'head' or beginning
    # of our result. We'll send just enough back so the client
    # can get the gist of our success or failure.
    #
    set x="`head $$.dat | sed s/\'/\'\'/g`"

    #
    # We are going to use SQL*PLUS only to interact with
    # the database. Here, we will feed SQL*PLUS a script
    # inline. We want the $x and $1 to be 'replaced' with
    # the corresponding environment variables, so we redirect
    # in from EOF. Later, we don't want this replacement
    # so we use 'EOF'.
    #
    # In this block of code, we simply write out the result
    # and send it on the pipe the client asked us to. The
    # pipe name is carried along as a parameter to this script.
    #
    # Note how to use CURSOR_SHARING=FORCE to make use of
    # bind variables. Since cursor sharing does not happen inside
    # of PL/SQL, we use a temporary table and insert our 'parameters'
    # into the temp table. PL/SQL reads those values out. This ensures
    # we do not overload our shared pool with all of the
    # character string literals we would otherwise. You need to have issued
    # create global temporary table host_tmp(msg varchar2(4000), pipe
    # varchar2(255) ); prior to executing this script.
    #
    SQL*PLUS / <<EOF
    alter session set cursor_sharing=force;
    insert into host_tmp (msg,pipe) values ( '$x','$1' );
    declare
        status      number;
        l_msg       varchar2(4000);
        l_pipe      varchar2(255);
    begin
        select msg, pipe into l_msg, l_pipe from host_tmp;
        dbms_pipe.pack_message( l_msg );
        status := dbms_pipe.send_message( l_pipe);
        if ( status <> 0 )
        then
          raise_application_error( -20001, 'Pipe error' );
        end if;
    end;
/
EOF

endif
```

```
#
# Now we go into 'server' mode. We will run and wait
# for something to be written on the pipe to us. We
# start by creating a private pipe. If one already exists,
# this does nothing. If one does not exist, the 'server' creates
# a private pipe for us.
#
# Then, we block indefinitely in received message, waiting for
# a request.
#
# Then we get the components of the message. Ours has 2 pieces:
# - the pipe to write back on
# - the command to execute
# We simply echo this out using dbms_output. If we receive
# any sort of timeout or error, we write out 'exit' which will
# cause us to exit.
#
# The command we run, SQL*PLUS / .... > tmp.csh is set up to do the
# following:
# SQL*PLUS /  -> run SQL*PLUS
# <<"EOF"     -> get the input inline, 'EOF' means don't do shell expansions
# | grep '^#' -> we'll keep only lines that begin with #
# | sed 's/^.//' -> get rid of that # in the real script
# > tmp.csh -> put that output from SQL*PLUS into tmp.csh
#
SQL*PLUS / <<"EOF" | grep '^#' | sed 's/^.//' > tmp.csh

set serveroutput on

whenever sqlerror exit

declare
        status      number;
        answer_pipe varchar2(255);
        command     varchar2(255);
begin
        status := dbms_pipe.create_pipe( pipename => 'HOST_PIPE',
                                         private  => TRUE );

        status := dbms_pipe.receive_message( 'HOST_PIPE');
        if ( status <> 0 ) then
                dbms_output.put_line( '#exit' );
        else
                dbms_pipe.unpack_message( answer_pipe );
                dbms_pipe.unpack_message( command );
                dbms_output.put_line( '##!/bin/csh -f' );
                dbms_output.put_line( '#' || command );
                dbms_output.put_line( '#exec host.csh ' || answer_pipe );
        end if;
end;
/
spool off
"EOF"

#
# Lastly, we make the script we just created executable
# and run it. We use EXEC in order to reuse the same pid.
# The last line in tmp.csh always re-runs this script using
```

```
# exec host.csh so we are in a loop of sorts...
#
chmod +x tmp.csh
exec tmp.csh >& $$.dat
```

That completes our simple server. It will get a command, run it and capture the output, send the head of the output back to the client, and wait for another command. We must run this simple server ourselves after starting the database in the background (and you may start more than one to allow for more than one host command at a time to be processed).

Now for the simple client, which will be a single small PL/SQL routine. The code for this simple client is:

```
ops$tkyte@ORA8I.WORLD> create global temporary table host_tmp
  2  ( msg  varchar2(4000),
  3    pipe varchar2(255)
  4  )
  5  /

Table created.

ops$tkyte@ORA8I.WORLD> create or replace procedure host( cmd in varchar2 )
  2  as
  3      status number;
  4      resp   long;
  5      answer_pipe varchar2(255)
  6                  default replace(dbms_pipe.unique_session_name,'$','_');
  7  begin
  8      -- Send to the 'server' a pipe to answer on and
  9      -- the command we would like to run.
 10      dbms_pipe.pack_message( answer_pipe );
 11      dbms_pipe.pack_message( cmd );
 12
 13      -- Send that message to the server.
 14      status := dbms_pipe.send_message( 'HOST_PIPE' );
 15      if ( status <> 0 )
 16      then
 17        raise_application_error( -20001, 'Pipe error' );
 18      end if;
 19
 20      -- Now, wait for the response back from the server.
 21      status := dbms_pipe.receive_message( answer_pipe );
 22      if ( status <> 0 )
 23      then
 24        raise_application_error( -20001, 'Pipe error' );
 25      end if;
 26
 27      -- Our 'protocol' says that the only thing
 28      -- the server will write back to us is the head
 29      -- of the output of our command.
 30      dbms_pipe.unpack_message( resp );
 31
 32      -- Here, we just 'print' the response out...
 33      dbms_output.put_line( substr( resp, 1, 80 ) );
```

```
 34  end;
 35  /

Procedure created.
```

Now that we have these two small pieces, we are ready to run host commands on the database server using PL/SQL. It might look like this:

```
ops$tkyte@ORA8I.WORLD> exec host( 'uptime' );
12:20pm  up 29 day(s),  8:06,  8 users,  load average: 2.71, 3.14, 2.97

PL/SQL procedure successfully completed.

ops$tkyte@ORA8I.WORLD> exec host( 'uname -a' );
SunOS aria 5.6 Generic_105181-23 sun4u sparc SUNW,Ultra-4

PL/SQL procedure successfully completed.
```

You must realize that in this simple example, the host command will be executed using the privileges of the user that started the `host.csh` script. Since we are using a private pipe, only the host PL/SQL routine can write on the `HOST_PIPE`, so we do not have to worry about rouge processes requesting us to execute arbitrary commands. Anyone with access to the `HOST` PL/SQL routine however, can run any command they want as that user on our system. Care must be taken to ensure only the commands you want to execute are in fact, executed. A safe implementation would have more logic in the `HOST` PL/SQL routine to inspect the command being executed, to ensure it is one of the commands you want to execute. A bulletproof implementation would not expose a `HOST` command at all, but rather expose only the host commands you wanted. For example, instead of granted `EXECUTE` on `HOST` to people, you might have a package:

```
create or replace package host_cmds
as
    function ls( p_directory in varchar2 ) return varchar2;
    function lpr( p_options  in varchar2, p_filename in varchar2 )
                 return varchar2;
    ...
end;
```

and grant `EXECUTE` on that package to people (keeping `HOST` as an internal, non-published routine).