Debugging DB2/CS client/server applications

by M. Meier
H. Pan
G. Y. Fuh

The technology of running external programs on the server side of a relational database management system (RDBMS) has been developed in the past few years. Database 2™/ Common Server (DB2™/CS) for UNIX™-based platforms supports external programs (i.e., user-defined functions and stored procedures) that are written by the application developer in a third-generation language such as C or C++. The main difficulty in debugging these external programs is that they are executed under the control of DB2/CS, which is itself a large software system for which no source code is provided. It is therefore impractical for a debugger to penetrate through the layers of software of DB2/CS to locate and debug the external programs. It is also very difficult for the debugger to determine when an external program will be invoked by the database engine and in which process it will be run. In addition, in an environment where the DB2/CS server is shared between a large number of users, it is necessary to ensure that the debugger does not violate the security of the DB2/CS system. In this paper, we describe a set of extensions to a distributed debugger and DB2/CS to support the debugging of external programs. A prototype was implemented to show the feasibility of the proposed approach.

The technology of running external programs on the server side of a relational database management system (RDBMS) has been developed in the past few years. For example, Database 2*/Common Server (DB2*/CS) for UNIX**-based platforms[1] supports external programs that are written by the application developer in a third-generation language (3GL) such as C or C++. There are currently two types of supported external programs:

1. A *user-defined function* (UDF) extends the functionality of DB2/CS by allowing users to define their own structured query language (SQL) functions implemented in a 3GL. Once created, a UDF can be invoked from any context where an SQL expression[2] is expected.

   As an example, let "payroll" be a table populated with the payroll information of a company. Execution of the following SQL query statement will run the user-defined function "under_paid" as part of the query on the server machine:

   ```
   SELECT empname FROM payroll WHERE
      under_paid(salary, education, experience) = 1;
   ```

2. A *stored procedure* allows the application developer to break a database application program into a client part and a server part. The server part can issue SQL requests while running on the same machine as the DB2/CS server. Results from the execution of the stored procedure can be passed back to the client part, which is usually running on a different machine. In some database applications, this can greatly improve performance.

0018-8670/97/$5.00 © 1997 IBM

Currently, there is no practical method for debugging these external programs. Although the current practice is to write a test driver program that simulates the DB2/CS call to the external program, external programs are executed under the control of the database engine, which is itself a large software system for which no source code is provided. It is therefore impractical for a debugger to penetrate through the layers of software of the database engine to locate and debug the external programs. It is also very difficult for the debugger to determine when an external program will be invoked by the database engine and in which process it will be run.

Another problem is that external programs are not statically linked with any executable module. Instead, when about to be invoked they are dynamically loaded by the database engine, which further complicates the situation for the debugger.[3] In addition, in an environment where the DB2/CS server is shared between a large number of users, it is necessary to ensure that the debugger does not violate the security of DB2/CS or the underlying operating system.

In this paper, we describe a set of extensions to a distributed debugger,[4,5] SQL, and the DB2/CS database engine to support the debugging of DB2/CS external programs. In our approach the user does not need to make any modification to the source code of the external programs. However, the external programs need to be recompiled with the compiler debugging option turned on (e.g., the "-g" option of the Advanced Interactive Executive* [AIX*] C compiler). In most cases, the user will add some additional SQL statements to the client program to activate debugging for the external programs and to set certain debugging options.

The extensions to the distributed debugger include a mechanism that allows the DB2/CS database engine to invoke a debugger library routine to request debugging services from a distributed debugger that may be running on a different machine. The distributed debugger can then "dynamically attach"[6] to the process that is running the external program.

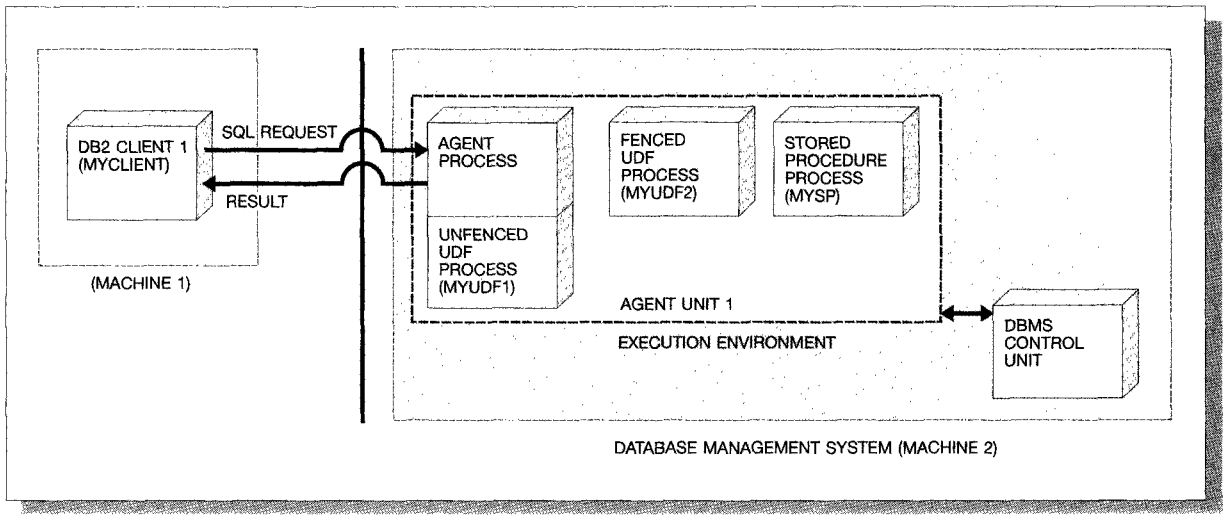
The debugger library routine will locate a distributed debugger that meets user-specified criteria and send it a message containing the information needed to locate the process running the external program, including the host ID (identifier) of the machine, the AIX process ID, and the thread ID. Also included is the information needed to obtain authorization for the debugger to attach to the process running the external program (i.e., login ID and password) and the instruction address in the external program where the debugging session should begin (e.g., the entry point of the external program).

The extensions also include proposed enhancements to the SQL standard and the database engine. Enhancements to the SQL standard are required to ensure that database security can be preserved in the presence of debugging support. This approach eliminates any possibility of circumventing the authority checking supported by the RDBMS. The proposed enhancements to SQL include a new "SET DBENV" statement that the user can invoke from a client program to set various environment variables. Some of these environment variables are used by the debugger library routine to locate a debugger in a distributed environment and to specify various debugger options. Also proposed is a new "DEBUG" SQL statement that can be invoked from the DB2/CS client program to indicate which external programs the database engine should request debugging services for, and under what conditions. In addition, we enhance the existing "GRANT" SQL statement with a new DEBUG privilege that can be used by an RDBMS system administrator to grant privileges for debugging a specified external program to other users. The "REVOKE" SQL statement is correspondingly enhanced.

In our approach the user does not need to modify the source code of the external programs.

The extensions proposed to the DB2/CS database engine allow the debugger to retrieve its internal state, at run time, as a set of data structures analogous to the caller stack of C. The data structures are maintained by the database engine in shared memory, accessible to the debugger, allowing it to determine, for example, the calling sequence (e.g., a DB2/CS client program invokes a stored procedure that in turn executes an SQL statement that executes a user-defined function, etc.).

Figure 1 Run-time environment of DB2/CS



The rest of the paper is organized as follows. In the next section we describe the external programs and the run-time environment of DB2/CS. Following sections introduce the architecture of a distributed debugger, describe extensions to the debugger, describe the extensions we propose to the SQL standard and the DB2/CS database engine, and summarize our debugging scenario. The final section contains our concluding remarks.

DB2/CS external programs and run-time environment

As mentioned earlier, there are two kinds of external programs: user-defined functions (UDFs) and stored procedures. In DB2/CS, a UDF can be run in two modes, *fenced* and *unfenced*. A fenced UDF provides better security, reliability, and data integrity at the expense of performance by creating a "firewall" between the database engine run-time code and the UDF run-time code. This is achieved by running UDF code in its own separate process. An unfenced UDF provides better performance at the expense of security, reliability, and data integrity by running the UDF code in the same process as the database engine.

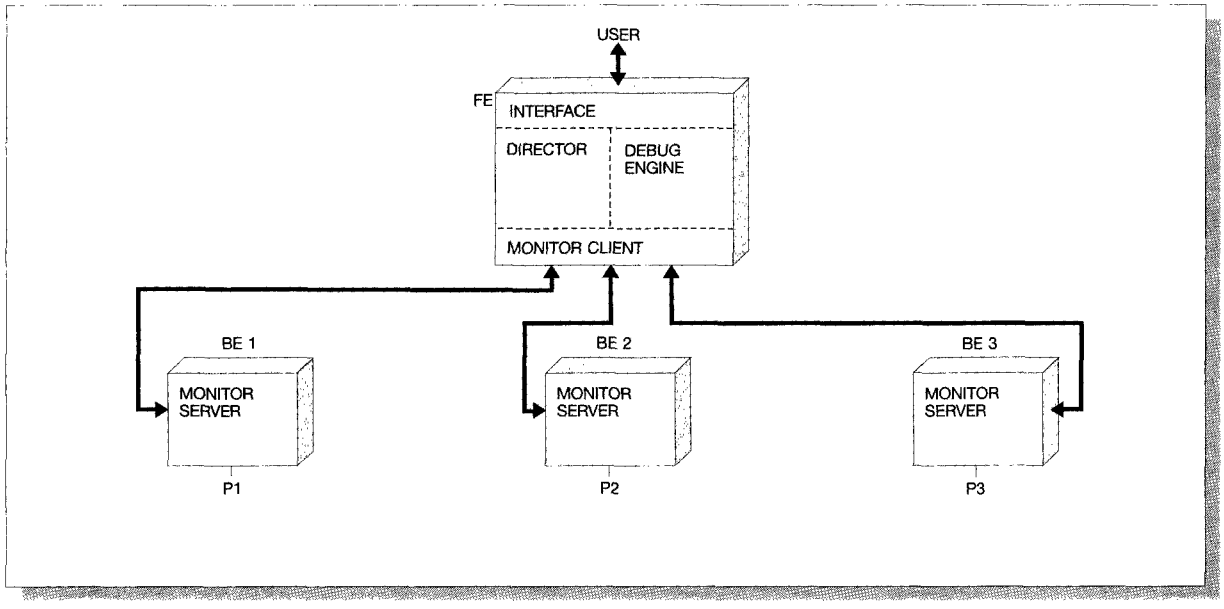
Figure 1 characterizes the DB2/CS run-time environment. A client program, "myclient," is running in a process on Machine 1. DB2/CS external programs (i.e., stored procedure and user-defined functions) are executed under the control of the DB2/CS server on Machine 2.

The DB2/CS server is a set of control processes, represented by the *DBMS control unit*[1] box, created at the same time as the database instance. The control unit "listens" for "connection requests" from clients. For each connection request, the control unit spawns a new set of processes referred to as an *agent unit*. The newly created agent unit is then connected to the corresponding client for receiving and serving the subsequent database requests. The agent unit consists of an *agent process* and, optionally, a set of *fenced UDF processes*, a set of *stored procedure processes*, or both.

Because it runs most of the database engine code, the agent process is also referred to as the "database engine." It receives the client request and distributes it to various service components to accomplish the requested action. In addition to the database engine code, the agent process runs unfenced UDF code.

A fenced UDF has its own process, which executes the run-time code to communicate with the agent process, dynamically loads the UDF library, and runs the fenced UDF code. There can be several UDF processes associated with an agent unit.[7] In many aspects, a stored procedure process has the same run-time characteristics as a fenced UDF process.

Figure 2 Architecture of a distributed debugger



However, unlike a fenced UDF process, a stored procedure process runs the external program as if it were a stand-alone application on the server machine.

A scenario using the components shown in Figure 1 follows. The client program "myclient" executes a stored procedure named "mysp." The stored procedure then executes an "SQL SELECT" statement that contains a call to an unfenced UDF named "myudf1." The body of "myudf1" contains an SQL statement that invokes the fenced UDF named "myudf2":[8]

```
mysp:
  SELECT empname FROM payroll WHERE
                        myudf1(salary, ssn) = 1;
```

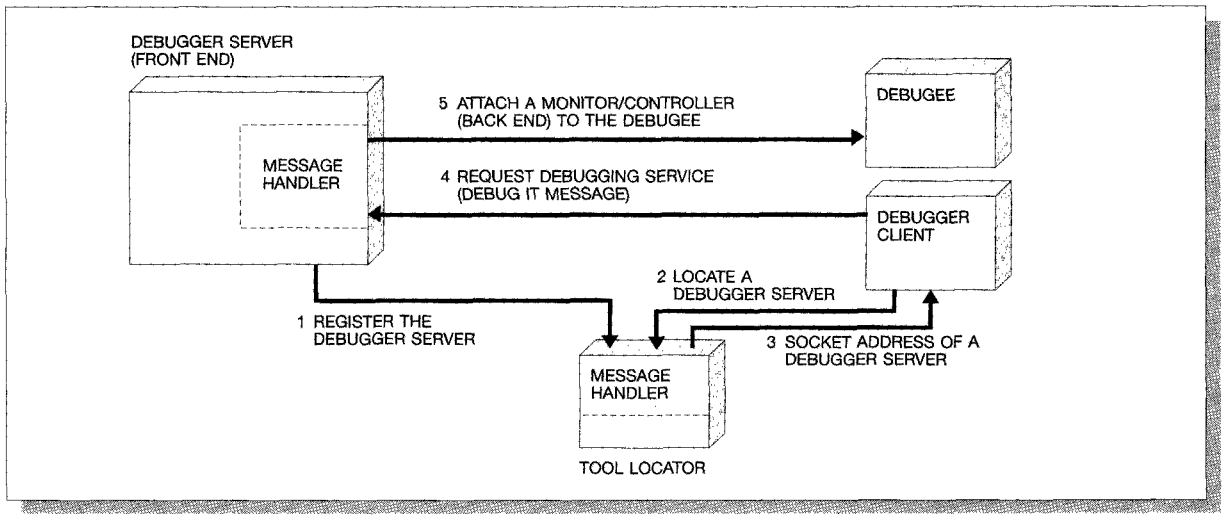
The database engine loads the stored procedure and the fenced UDF, in their own separate processes, on the machine that is running the DB2/CS server (Machine 2). The unfenced UDF is loaded into the same process as the database engine.

Architecture of a distributed debugger

Figure 2 shows the architecture of the prototype distributed debugger "Parallel and Distributed Dynamic Analyzer" (PDDA),[5] which is the basis of our work described in this paper. PDDA was originally developed to debug parallel and distributed applications that use Open Software Foundation's Distributed Computing Environment** (OSF DCE**).[9] It is based on a prototype version of the AIX *xldb*[10] single process debugger. Xldb, an IBM program product, is based on the X Window System**. Using the McDowell-Helmbold classification of basic approaches for debugging concurrent programs,[11] PDDA can be categorized as an extension of traditional debugging techniques (breakpoints set by users) for parallel and distributed programs. PDDA has recently been extended to support debugging for IBM Distributed System Object Model (DSOM)[12] applications.[4] The architecture of PDDA includes a front end (FE) and one or more back ends (BEs) attached to processes (P1, P2, P3) running the application programs that use the underlying debugging support of the operating system (e.g., the AIX/6000* "ptrace" function).

The front end provides a single user interface and handles most of the initialization and parallel execution control issues. Moreover, it creates a back end for each program involved in the application. The back end runs on the same host as the application program and will carry out requests from the front end to monitor and control it. These requests include reading and writing the program state, starting and

Figure 3 Dynamic connection to a debugger in a distributed environment



stopping the execution of the program, and monitoring the program for interrupts (e.g., breakpoints, floating point exceptions, etc.).

A back end can be created either during debugger initialization or dynamically during the debugging session, using the "dynamic connection" approach described in the next section. When the back end is created during debugger initialization, the program to be monitored and controlled must be specified. When the back end is created dynamically, the program to be monitored and controlled is *not* specified at debugger initialization; instead the application program calls a debugger library routine to request debugging services from a particular instance of a distributed debugger. An application program can request debugging services for itself or for another program.

Extensions to the distributed debugger

Although intuitively it seems that the debugging of an external program should be under the control of the DB2/CS client program that invokes it, there are three obstacles.[3] The first is a timing obstacle. The DB2/CS external program's process is created on demand, and the external library is loaded and unloaded dynamically. The next is an authorization obstacle. A DB2/CS process usually runs under a special user ID so that normal users cannot attach it. Finally, there is an access obstacle. In general a remote user does not have a user account on the DBMS server machine, making it extremely difficult, if not impossible, to debug an external program's process from the client machine.

These three obstacles may appear to be orthogonal. However, as we examined them carefully we discovered that they all originate from the same source: the expectation that the debugger is to be initiated from the DB2/CS client program. We then started to think from an entirely different perspective: suppose the DBMS initiated the debugger? This would overcome the timing obstacle, the authorization obstacle, and the access obstacle.

The obstacles were then reduced to the problem of how to initiate a debugging session from the DBMS. To solve this problem we have developed a general-purpose facility, the "Dynamic Connection"[13] component. This component allows an application program to locate a distributed debugger front end in a distributed environment at run time and send it a message requesting debugging services. The debugger then attaches a back end (monitor/controller) to the process for which debugging services were requested. The DBMS can use the "Dynamic Connection" component to initiate a debugging session for an external program.

Dynamic connection overview. Figure 3 illustrates a dynamic connection to a debugger in a distributed

environment. Before describing it, we define some of the terms used. A *debugee* is a program that is to be debugged (e.g., a DB2/CS external program). A *debugger server* is a program that provides debugging services (i.e., the distributed debugger front end). A *debugger client* is a program (e.g., the DB2/CS database engine) that sends a request to a debugger server to provide debugging services for a debugee. The debugee can be the debugger client itself, or another program running anywhere on the network. A *monitor/controller* is a program that is attached to the debugee, by the debugger server, to monitor and control its execution and to read and write state information (i.e., the distributed debugger back end).

The debugger client calls a debugger library routine to locate a debugger server and send it a message that contains the information needed to locate the debugee and obtain authorization to attach a monitor/controller. Additionally, the message will include the instruction address in the debugee where the user would like the debugging session to begin (e.g., at the current instruction address of the debugee or at entry to a routine invoked by the debugee program).

The debugger server may or may not be running on the same machine as the debugee. To locate the debugger server, the *tool locator*, a new component, is used. The debugger client and debugger server can communicate with the tool locator through *socket* connections.[14]

Figure 3 shows the sequence of steps for a debugger client (e.g., the DB2/CS database engine) to dynamically request debugging services for another program (e.g., a DB2/CS external program). (1) A debugger server (i.e., a distributed debugger front end) is started and registers itself with the tool locator, indicating that it is available to serve debugging requests. (2) A debugger client sends a message to the tool locator to locate a debugger server. (3) The tool locator returns the socket address of a debugger server that matches the debugger client's specification. (4) The debugger client sends a "debug it" message to request debugging service for the debugee from the debugger server. (5) The debugger server attaches a monitor/controller to the debugee.

Tool locator. The tool locator is a general-purpose mechanism for first registering, then locating programs that have certain properties in a distributed environment. The DB2/CS database engine as debugger client can call a debugger library routine that will use the tool locator to find a registered distributed debugger front-end server with certain properties, such as the user ID it is running under, the machine it is running on, the X Window System display it is using, the programming languages and operating systems it supports, etc.

When a distributed debugger front end is started, it calls a debugger library routine to register with the tool locator, passing to it a string that contains *property-name* = *property-value* substrings separated by commas. For example, the string

```
"hostname=atlantic,userid=hpan,opersys=AIX,
                        language=C,language=CPP"
```

indicates that the distributed debugger is running on a host named "atlantic" under the user ID "hpan" and supports the debugging of programs written in C and C++ on AIX. DB2/CS can then execute a debugger library routine, specifying as one of its arguments search criteria for a debugger front end running under the user ID "hpan" that supports C++ on AIX.

The search criteria argument is a string that contains conjunctions and disjunctions of *property-name* = *property-value* expressions. Parentheses can be used to specify precedence. There are no predefined property names or property values; these are simply arbitrary sequences of case-insensitive alphanumeric characters. For example, the string

```
"userid=hpan and machtype=rs6000 and
 opersys=AIX and language=C and language=CPP"
```

could be used to locate any debugger front end running on any host under user ID "hpan" that supports the debugging of C and C++ programs running in AIX on a RISC (reduced instruction-set computer) System/6000**.

As another example, the string

```
"opersys=WindowsNT and language=CPP and
                              machtype=PowerPC and
  ((userid=hpan and hostname=davinci) or
   (userid=meier and hostname=atlantic) or
   userid=fuh)"
```

could be used to locate any distributed debugger front end that supports programs written in C++ for Windows NT** on a PowerPC* that is either running under user ID "hpan" on a host named "davinci"

or running under user ID "meier" on a host named "atlantic" or running under user ID "fuh" on any host.

The tool locator returns to the library routine a socket address for the first debugger front end that matches the criteria. (If more than one debugger front end matches the search criteria, their socket addresses can be retrieved by subsequently executing a series of *FindNext* calls to the tool locator.) The library routine will use the socket address to create a socket connection and send a message to the debugger front end, requesting it to attach a monitor/controller to the process that is running the external program.

Debugger client application program interface. Two application programming interface (API) routines are provided by the debugger library for the debugger client to request debugging services. If the debugger client is also the debugee, it calls "debugMe." If another program is to be debugged, the client calls "debugIt."

The debugIt routine. A call to the debugIt routine requires a search criteria used to locate a distributed debugger front end (i.e., the debugger server). The arguments include all of the information needed to locate a particular application program and attach a debugger back end (i.e., monitor/controller) to it:

```
void debugIt(char   *searchcriteria,
  char *netaddr,
  char *userid,
  char *password,
  int   addrspaceid,
  int   threadid,
  unsigned int instraddr;
  char *dbgservargs[256],
  int   *status);
```

where:

searchcriteria is a string that contains the search criteria described earlier. If NULL is specified then the current value of the DEBUGSEARCHCRITERIA environment variable will be used.

netaddr is the network address of the machine where the debugee is running.

userid is the user ID that the debugee is running under.

password is the password of the user ID that the debugee is running under.

addrspaceid is the address-space ID (i.e., UNIX process ID) that the debugee is running under.

threadid is the thread ID that the debugee is running under.

instraddr is the instruction address where the debugging session should begin. If *zero* is specified, then the current instruction address of the debugee is used.

dbgservargs is a string of up to 256 characters (including the terminating NULL) that can contain options for the debugger (e.g., where to find source code).

status is a pointer to an integer where the status code is returned. The value returned is one of:

error_status_ok Normal completion

dbg_no_debugger_server_found No debugger server found for specified search criteria

dbg_debugger_server_reject Request rejected

For example:

```
debugIt ("userid=hpan and machtype=rs6000 and
                    opersys=AIX and language=CPP",
          "thistle.stl.ibm.com", "meier", "mypasswd",
                                          35647, 1, 0,
          "-s /u/hpan/code -s /u/hpan/src", &status);
```

will cause the debugIt routine to:

1. Look for an environment variable named TOOLLOCATORHOST that specifies the host name of the tool locator.
2. Execute a call to the BeginSearch debugger library routine, passing the search criteria. The BeginSearch routine will return a socket to be used to communicate with the tool locator.
3. Execute a call to the FindNext debugger library routine, which, if successful, returns the socket address of the next corresponding debugger server. If unsuccessful, debugIt returns to the caller, passing back the error status.
4. Use the socket address returned by the FindNext routine to establish a connection to the distributed debugger front end (i.e., the debugger server) and send it a "debug it" message that includes all of the arguments passed to the debugIt routine: network address, login ID, password, address-space ID, thread ID, instruction address, and the debugging options for the debugger front end.
5. Receive from the distributed debugger front end an acknowledgment message. If a negative acknowledgment is received or a time-out occurs, then debugIt repeats the previous two steps until a status is returned from FindNext indicating that there are no more distributed debugging front ends that meet the criteria.

6. Execute a call to the EndSearch debugger library routine to end the search session and close the connection to the tool locator

The debugMe routine. A call to the debugMe routine requires the same search criteria as the debugIt routine. In this case, the debugger client is the debugee. The debugMe routine calls the debugIt routine, passing the search criteria along with the current network address, login ID, password, address-space ID, thread ID, and an instruction address where the debugging session should begin, which in this case is the return address of the debugMe call.

```
void debugMe (char *searchcriteria,
              char *dbgservargs[256],
              int  *status);
```

where:

searchcriteria is a string that contains the search criteria described earlier. If NULL is specified then the current value of the DEBUGSEARCHCRITERIA environment variable will be used.

dbgservargs is a string of up to 256 characters (including the terminating NULL) that can contain debugging options for the debugger (e.g., where to find source code).

status is a pointer to an integer where the status code is returned. The value returned is one of:

- error_status_ok Normal completion
- dbg_no_debugger_server_found No debugger server found for specified search criteria
- dbg_tool_locator_failure Call to the tool locator failed
- dbg_no_tool_locator_found No tool locator found

For example:

```
debugMe ("userid=hpan,machtype=rs6000,
                    opersys=AIX,language=CPP",
         "-s /u/hpan/code", &status);
```

calls the debugIt routine, passing the current network address, login ID, password, address-space ID, thread ID, and the return address of the debugMe routine as the instruction address where the debugging session should begin. The string "-s /u/hpan/code," which is the actual parameter for the variable dbgservargs, will be used by the distributed debugger front end (i.e., the debugger server) to locate the source code of the external programs.

Debugger server message handler. The distributed debugger front end (i.e., the debugger server) has been extended to "listen" for socket connection requests from the DB2/CS database engine (or any other debugger client). Once a connection is established, the distributed debugger front end will receive a "debug it" message from the DB2/CS database engine.

The "debug it" message contains the network address, login ID, password, address-space ID, thread ID, instruction address of a debugee where the debugging session should begin, and a debugging options string. The distributed debugger front end will parse the debugger options string to set the options, such as where to locate the source code, and then attach a monitor/controller to the DB2/CS external program (i.e., the debugee). A breakpoint is set at the instruction address where the debugging session is to begin and an acknowledgment message is sent back to the DB2/CS database engine. The debugger will then continue the execution of the DB2/CS external program until it encounters the breakpoint.

Figure 4 shows the components of Figure 1 running under the control of a distributed debugger with the dynamic connection extensions and the tool locator.

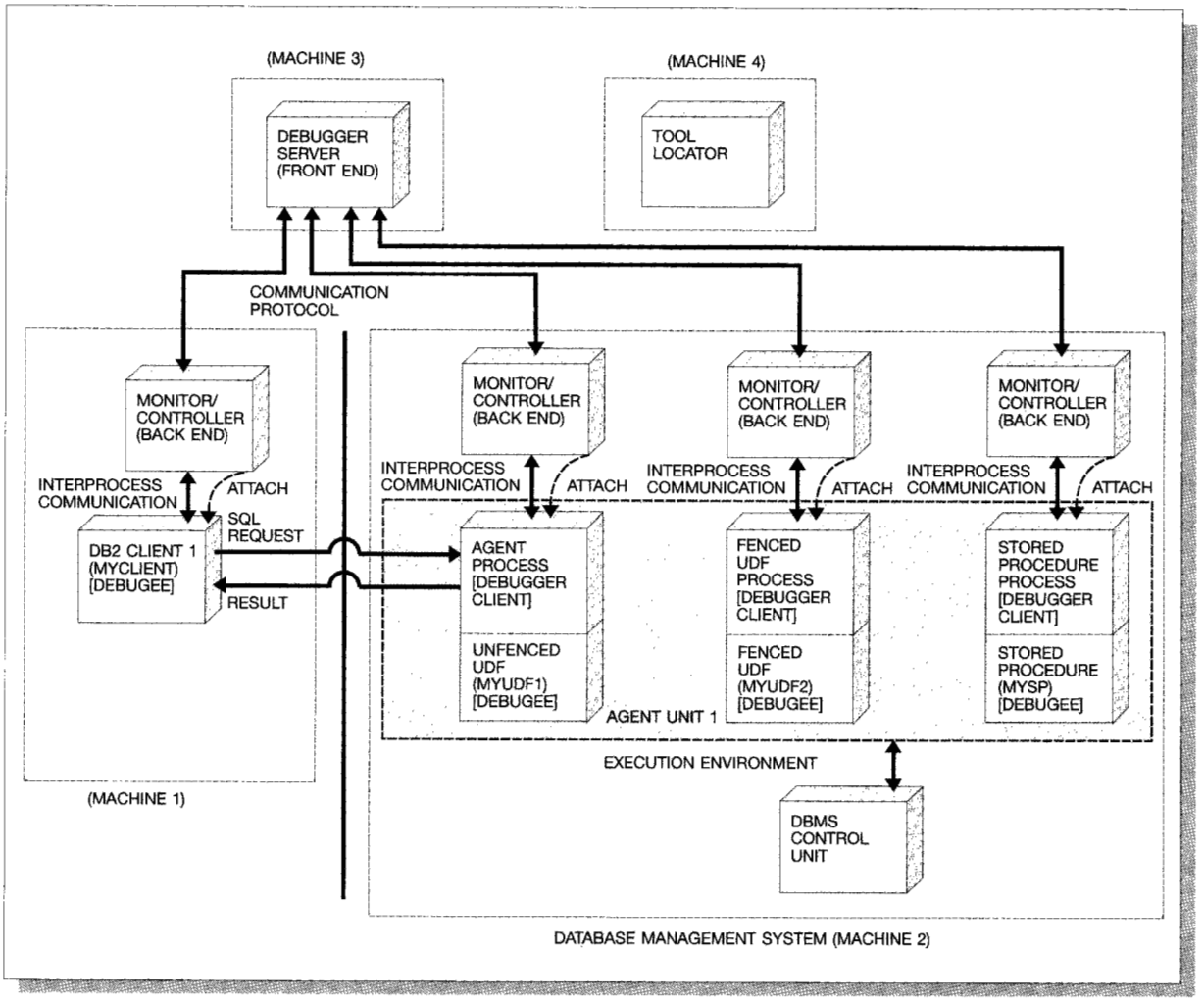
Extensions to DB2/CS

The extensions to the DB2/CS system include enhancements to both the database engine and to SQL.

Invocation stack frame. In order to access the internal state of the DB2/CS server, the DB2/CS agent process (i.e., the database engine) will maintain a stack of record structures, called *invocation stack frames* (ISFs), in shared memory. Each ISF record represents an invocation of a client program or an external program and includes the host machine and the process ID where the program is running, the user ID the program is running under, the entry point of the external program, etc. An ISF is analogous to an activation record in a call stack of a 3GL. The most recently invoked client program or external program is represented by the ISF record at the top of the stack.

Figure 5 represents the invocation stack for the example shown in Figure 4. In this case the invocation stack contains four records: the top record represents the fenced UDF function "myudf2," the next the unfenced UDF "myudf1," the next the stored procedure "mysp," and the bottom record represents the client program "myclient." The debugger front end can send a command to the back end that will execute a debugger library routine named

Figure 4 Debugging environment of a typical DB2/CS application



getInvStkFrame to read the ISF records from shared memory and pass them back to the front end.

SQL extensions. We propose new commands to be added to SQL to specify various debugging options and to enable debugging for a specified set of DB2/CS external programs.

The following grammar rules[15] define the syntax of the "SET DBENV" command. This command specifies various options used by the debugMe and debugIt routines to locate the debugger front end and to set debugging options:

```
set_dbenv_stmt:
  SET DBENV environment_attr_list ;

environment_attr_list :
  environment_attr |
  environment_attr_list , environment_attr ;

environment_attr :
  SEARCHCRITERIA = string  |
  TOOLLOCATORHOST = string  |
  SOURCE = string  |
  OPTIONS = string ;
```

For example, the following command:

```
SET DBENV SEARCHCRITERIA = 'userid=fuh and
                    language=C and opersys=AIX',
          TOOLLOCATORHOST =
                              'bigblue.stl.ibm.com',
          SOURCE =
                    '/u/fuh/udf_src:/u/hpan/sp_src',
          OPTIONS = '-T /u/fuh/mytmpdir';
```

will direct the debugMe and debugIt library routines to use the instance of a distributed debugger that has registered with the tool locator running on the host named "bigblue.stl.ibm.com" with properties that indicate it is running under the user ID "fuh" and supports the debugging of programs written in C on AIX. The debugger will search directories /u/fuh/udf_src and /u/hpan/sp_src for the source code of the external programs. The options "-T /u/fuh/mytmpdir" will be passed to the debugger by the debugMe and debugIt routines.

A new SQL "DEBUG" statement specifies which external programs should be debugged and under what conditions. The following grammar rules apply:

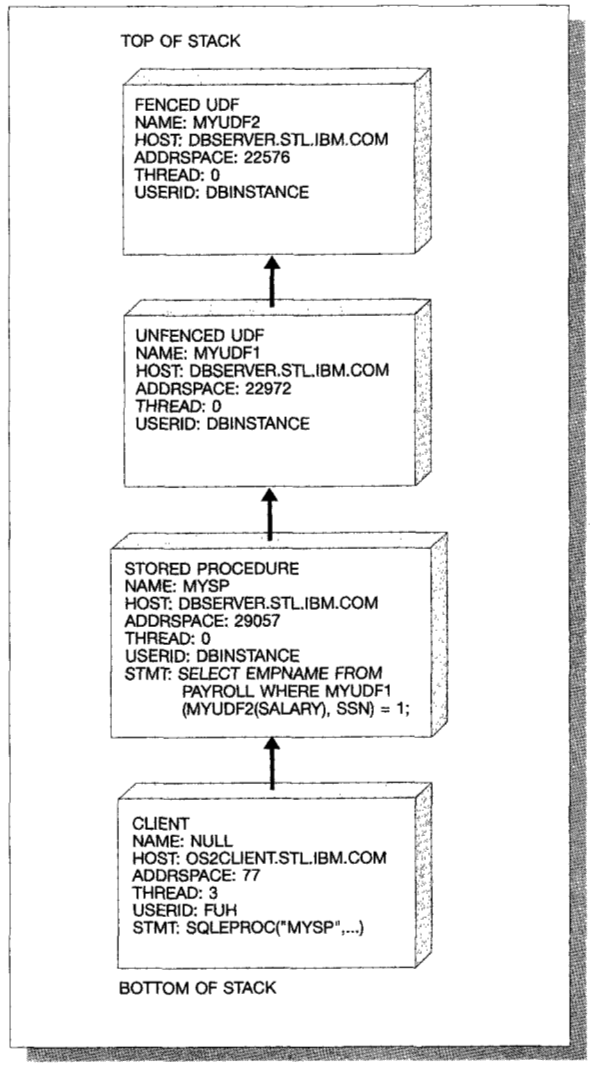
```
debug_stmt :
  DEBUG debug_intent program_ref_clause
                                  debug_condition ;

debug_intent :
  ON |
  OFF ;

program_ref_clause :
  STORED PROCEDURE sp_ref_list |
  FUNCTION function_ref_list ;

debug_condition :
                  |
  WHEN parameter_exp |
  AT int_constant CALL;
```

In these grammar rules, *parameter_exp* represents a Boolean function defined over the formal parameters of the associated UDF, and *int_constant* is an integer literal specifying the iteration in which the associated UDF is invoked in the current statement. A *debug_condition* cannot be specified for a stored procedure.

The following examples demonstrate the use of the "DEBUG" command:

- Each of the selected functions will be debugged each time it is executed:

Figure 5 Example of an invocation stack containing four invocation stack frames (ISFs)




```
DEBUG ON FUNCTION udf_integer, udf_float ;
```

- Each of the selected functions will be debugged if its first parameter is equal to 0:

```
DEBUG ON FUNCTION udf_integer, udf_float
                                  WHEN #1 = 0 ;
```

- The selected function will be debugged the first time it is invoked in a statement:

```
DEBUG ON FUNCTION my_udf AT 1 CALL ;
```

- The selected stored procedure will no longer be debugged:

```
DEBUG OFF STORED PROCEDURE
                        /u/fuh/sp/sp_lib/my_sp ;
```

In addition, an enhanced SQL "GRANT" statement specifies which users are allowed to debug a selected external program. The following grammar rules apply:

```
debug_stmt :
  GRANT DEBUG ON FUNCTION function_ref_list
                                TO authid_list |
  GRANT DEBUG ON PROCEDURE sp_ref_list
                                TO authid_list ;

authid_list :
  authid_item |
  authid_list , authid_item ;

authid_item :
  authid |
  PUBLIC ;
```

For example, the following command:

```
GRANT DEBUG FUNCTION my_udf
                        TO hpan, meier, fuh ;
```

will grant authority to debug the user-defined function "my_udf" to the DB2/CS authorization IDs "hpan," "meier," and "fuh."

And finally, the SQL "REVOKE" statement has an associated new DEBUG privilege to revoke the debugging capability of a selected external program from other users:

```
debug_stmt :
  REVOKE DEBUG ON FUNCTION function_ref_list
                              FROM authid_list |
  REVOKE DEBUG ON PROCEDURE sp_ref_list
                              FROM authid_list ;
```

For example, the following command:

```
REVOKE DEBUG FUNCTION my_udf
                        FROM hpan, meier, fuh ;
```

will revoke the authority to debug the user-defined function "my_udf" from the DB2/CS authorization IDs "hpan," "meier," and "fuh."

The control of debugging activities is fully integrated into the underlying DB2/CS. Several advantages are offered by an integrated debugging environment:

- Debugging control is specified using SQL commands; no changes need to be made in external programs. Thus they require neither recompilation nor relinking to turn debugging on and off.
- Conditional debugging, specified in terms of the SQL context, can be efficiently supported. It would be very difficult, if not impossible, to support this feature without integration between the debugger and DB2/CS.
- Since debugging activity is controlled by DB2/CS, the authority checking currently supported by DB2/CS can be easily extended to control the debugging requests. Therefore, database security can be preserved in the presence of debugging support.

Debugging scenario

The following is a summary of steps that are performed by the DB2/CS application program (including the client program and external programs), DB2/CS database engine, and the distributed debugger for the example shown in Figures 4 and 5.

1. The user starts the debugger front end, specifying that "myclient," a DB2 client program, is to be debugged.

2. The debugger front end starts "myclient" running, under control of a debugger back end.

3. The program "myclient" first executes the SQL "CONNECT" command, then the SQL "SET DBENV" command, specifying environment variables to set debugging options, such as which debugger front end to use and where to find the source code for the external programs used by "myclient." (The DB2 client program probably would have been coded to request the same debugger front end that the user has already started.)

4. The program "myclient" executes SQL "DEBUG" commands to indicate that the external programs "mysp," "myudf1," and "myudf2" are to be debugged.

5. When the agent process (the DB2 database engine) receives the SQL request to begin debugging, DB2/CS creates an ISF record for the client program and fills in the fields. This record is then

pushed onto the invocation stack as the first entry. Note that the agent process is the debugger client.

6. When the agent process determines that it is about to invoke an external program that requires debugging services, it creates an ISF record, filling in all the fields, and adds it to the invocation stack.

7. Just before the external program is to be executed, DB2/CS calls a library routine named debuggerBeginExtProg, located in the process where the external program will run. The syntax is:

   ```
   void debuggerBeginExtProg(int *status);
   ```

   where:

   status is a status code with values:
   error_status_ok No errors
   cannot_start_debugger Unable to start the debugger

   The routine checks to determine whether or not a debugger back end is currently attached. If not, it sends a request to the debugger front end to attach a debugger back end to the external program process.

   The debuggerBeginExtProg routine then executes a breakpoint instruction to signal the debugger back end that an external program is about to be executed. The debugger back end in turn notifies the debugger front end. The breakpoint instruction causes the external program process to suspend its execution until the debugger front end issues a "continue execution" command.

8. When the debugger front end receives notification that an external program is about to be executed, it sends a command to the debugger back end, which calls the getInvStkFrame debugger library routine to get the ISF record at the top of the stack and sends it back to the front end in a reply. This record represents the external program that is about to be executed.

9. The debugger front end gets the entry point of the external program from the ISF record and sets a breakpoint there. It then executes a "continue execution" command for the external program process.

10. The external program immediately encounters the breakpoint.

11. The debugger then uses the getInvStkFrame routine to get the information it needs to determine the complete context for the executing program, and displays the corresponding information to the user.

    For example, when "myudf2" is invoked, a distributed call stack[4] could be displayed showing that the "myclient" program called the "mysp" stored procedure, which in turn executed an SQL command that called the "myudf1" unfenced UDF, which called the fenced UDF "myudf2." The user could then click on any of the items in the distributed call stack to view the current state of the client program, stored procedure, or UDF.

12. When the external program returns, a call is made to debuggerEndExtProg by DB2/CS. The syntax is:

    ```
    void debuggerEndExtProg(int *status);
    ```

    where:

    status is a status code with values;
    error_status_ok No errors
    cannot_signal_debugger Unable to signal the debugger

    As described for the debuggerBeginExtProg routine, debuggerEndExtProg executes a breakpoint instruction to signal the debugger back end, which in turn notifies the debugger front end that an external program has just ended.

13. The debugger front end then executes getInvStkFrame to get the ISF record at the top of the stack. This record represents the external program that just ended.

14. If the external program is a stored procedure, it will detach the debugger back end from the external program process. This detach action is necessary, because DB2/CS assigns an already-created process to run a stored procedure. The next stored procedure run in that process may be for a different client. The agent process and fenced UDF processes are not shared among clients. For them, the debugger back end can stay attached and the debugger front end simply executes a "continue execution" command.

Concluding remarks

Now that DB2/CS has the ability to execute external programs such as stored procedures and user-defined functions, it is necessary to find a way to effectively and efficiently debug these programs. We have proposed a solution that provides a comprehensive debugging environment for DB2/CS client/server applications. The solution involves extensions to a distributed debugger, the SQL standard, and the DB2/CS database engine. Based on feasibility prototypes we have developed, we believe that the general approach can also be applied to debugging Customer Information Control System for the RISC System/6000 (CICS/6000*)[16] distributed transaction applications and Messaging and Queuing Series for the RISC System/6000 (MQSeries*)[17] distributed messaging applications.

Acknowledgments

This work would not have been possible without the continuing, enthusiastic, and inspiring support of Vivek Sarkar, manager of IBM's Application Development Technology Institute. We would also like to acknowledge Len Lyon, Brian Tran, Jyh-Herng Chow, and the anonymous referees for their many thoughtful comments.

Cited references and notes

1. *DATABASE 2 Application Programming Guide for Common Server*, S20H-4643-01, IBM Corporation (1995); available through IBM branch offices.
2. *DATABASE 2 SQL Reference for Common Server*, S20H-4665-01, IBM Corporation (1995); available through IBM branch offices.
3. G. Fuh, K. Nomura, M. Meier, H. Pan, and G. Wilson, "Debugging User-Defined Functions in RDBMS Client-Server Environment," *Proceedings of the 1996 International Computer Symposium*, Taiwan, December 19–21, 1996. (A previous version was released as IBM Technical Report ADTI-1994-020 [September 1994]; available by request from vndoadti@vnet.ibm.com.)
4. M. S. Meier, K. L. Miller, D. P. Pazel, and J. R. Rao, "Experiences with Building Distributed Debuggers," *Proceedings of the Symposium on Parallel and Distributed Tools*, Philadelphia, PA, May 22–23, 1996, pp. 70–79.
5. M. Meier, H. Pan, B. Harding, L. Lyon, and L. Scarborough, "Parallel and Distributed Dynamic Analyzer (PDDA)—A Debugger for Client/Server Programs," IBM Technical Report ADTI-1994-003 (July 1994); available by request from vndoadti@vnet.ibm.com.
6. The AIX/6000 "ptrace" function provides the underlying support that allows a debugger to monitor and control a second process. One of the functions provided by "ptrace" allows a debugger to "attach" to a running process to monitor and control its execution. See *Calls and Subroutines Reference for RISC System/6000*, SC23-2198-00, IBM Corporation (1990); available through IBM branch offices.
7. DB2/CS Version 2 and Version 3 currently allow only one UDF process associated with an agent unit. However, our proposed approach is designed to support multiple UDF processes.
8. DB2/CS Version 2 and Version 3 do not currently allow SQL statements to be executed from a UDF. However, our proposed approach is designed to support this.
9. Open Software Foundation, *Introduction to OSF DCE*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1992).
10. *User's Guide for C Set ++ Version 3.1 for AIX*, SC09-1968-01, IBM Corporation (1995); available through IBM branch offices.
11. C. E. McDowell and D. P. Helmbold, "Debugging Concurrent Programs," *ACM Computing Surveys* **21**, No. 4, 593–622 (December 1989).
12. *SOMobjects: A Practical Introduction to SOM and DSOM*, GG24-4357-00, IBM Corporation (1994); available through IBM branch offices.
13. M. Meier and H. Pan, "Dynamic Connection to a Debugger in a Distributed Environment," IBM Technical Report ADTI-1995-005 (June 1995); available by request from vndoadti@vnet.ibm.com.
14. W. R. Stevens, *UNIX Network Programming*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1990).
15. In this grammar, uppercase words are "terminal" symbols. Lowercase words are "nonterminal" symbols. Each rule starts with a nonterminal symbol followed by a colon, contains one or more alternative definitions for the symbol, and ends with a semicolon. Definitions contain terminal and nonterminal symbols; alternative definitions are separated by "|" (or). A command is formed from its defining rule by recursively replacing each nonterminal symbol with its definition.
16. *CICS/6000 Technical Overview*, GC33-1225-00, IBM Corporation (1993); available through IBM branch offices.
17. *MQSeries Concepts and Architecture*, GC33-1141-01, IBM Corporation (1994); available through IBM branch offices.

Accepted for publication September 16, 1996.

Mike Meier *IBM Software Solutions Division, P.O. Box 49023, San Jose, California 95141 (electronic mail: msmeier@vnet.ibm.com).* Mr. Meier is a senior software engineer in the IBM Application Development Technology Institute. His B.S. degree in mathematics was awarded by Lawrence Technological University, Southfield, Michigan. His more than 20 years of programming experience includes database, on-line teleprocessing (OLTP), expert systems, logic programming, semantic networks with object-oriented extensions, an object-oriented framework for scheduling applications, and debugging tools for parallel and distributed programs. Mr. Meier received an Outstanding Technical Achievement Award for his work in expert systems, and he holds a number of patents in the area of distributed debugging.

Hsin Pan *IBM Software Solutions Division, P.O. Box 49023, San Jose, California 95141 (electronic mail: hpan@vnet.ibm.com).* Dr. Pan is an advisory software engineer in the Application Development Technology Institute. He received the Ph.D. degree in computer science from Purdue University, West Lafayette, In-

diana, in 1993. His primary interest is to develop techniques and tools to assure the software quality, reliability, and safety for both structured and object-oriented programs. After joining IBM in 1993, he worked on the parallel and distributed debugger and the distributed computing objects. From August 1995 to July 1996 he was an associate professor in the Department of Computer and Information Science at National Chiao Tung University, Taiwan. He has been awarded a number of patents recognizing his work at IBM. He is a member of the ACM (Association for Computing Machinery), the IEEE (Institute of Electrical and Electronics Engineers), and the IEEE Computer Society.

Gene (You-Chin) Fuh *IBM Software Solutions Division, P.O. Box 49023, San Jose, California 95141 (electronic mail: fuh@vnet.ibm.com).* Dr. Fuh received a B.S. degree in computer science from National Taiwan University in 1981, and M.S. and Ph.D. degrees in computer science from the State University of New York at Stony Brook in 1986 and 1989. Since then, he has worked in the area of compiler development for various computer languages, such as VHDL (IEEE 1076 VHSIC [Very High Speed Integrated Circuit] Hardware Description Language), Verilog, FORTRAN 90, and SQL. He is currently one of the leaders in the Object Strike Force team whose mission is to develop new object-relational technologies for future releases of DB2/CS. Prior to joining IBM in 1993, Dr. Fuh held several technical management positions in the electronic CAD (computer-aided design) industry. His current technical interests are compiler construction, language design, object-relational database, client/server debugging methodology, and internet application development.

Reprint Order No. G321-5636.