LDAP Triggers: A Framework for Sun Java System Directory Server

By Nicola Venditti, SunPS, Italy

Sun BluePrints™ OnLine—February 2004

http://www.sun.com/blueprints

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95045 U.S.A.
650 960-1300

Part No. 817-5231-10
Revision 1.0, 2/11/04
Edition: February 2004

Please
Recycle

Adobe PostScript

LDAP Triggers: A Framework for Sun Java System Directory Server

This article describes how to implement SQL-like triggers in a Sun Java™ System Directory Server. The example scenario shows how to extend the server using the Plug-in API. This article is primarily directed at expert developers and architects who want to understand issues related to developing and deploying the Sun Java System Directory Server extension, implemented with plug-ins and extended operations.

This article contains the following topics:

Background

Employing triggers on database servers is useful when you need to define actions based on user operations or events. Through triggers, you can enforce rules such as: "when this table changes, do this," "before my data is deleted, do this other action," "when the user logs in/or updates some special record," and so on.

For example, the Customer Care System (which only knows of its own database, most often an Oracle database) deletes a customer record in the database after the customer unsubscribes from services or terminates a contract. Using triggers, you can implement an application to remove any trace of customer data from the Sun Java System Directory Servers. Generally speaking, through triggers you can reflect modifications on a central database to other data repositories, which have a copy of the data and must be kept in sync with main database. This is what the Sun Java System Meta-Directory does to keep track of changes in an ORACLE database; to capture changes on the Sun Java System Directory Server, however, Sun Java System Meta-Directory uses a different mechanism called `changelog`.

Unfortunately, LDAP lacks a standard mechanism to tell the LDAP server to trigger an action based on a user's operation. Sun Java System Directory Server 5.x has features that allow you to trigger an action using pre- and post-operation plug-ins. However, the server simply calls your plug-in function when a data operation of the kind you registered your plug-in for either happens or is going to happen. The rest is your implementation.

The code example presented in this article not only uses pre- and post- operation plug-ins, but takes it further by showing LDAP administrators how to create and manipulate triggers in a SQL style, using SQL-like instructions. To implement this last feature, you need the following, in addition to the pre- and post- plug-ins:

- Language specification
- Parser
- Extended operation

The language specification is required at compilation time for the parser generator to generate the parser itself. The parser is a module of generated C code that we embed into the extended operation function. The extended operation is required to allow users, through a client application, to submit their trigger commands to the Sun Java System Directory Server.

The code example is based on the Sun Java System Directory Server 5.2 (the newest release), and is based on UNIX® platforms. (The parser generator is available on most UNIX platforms.)

For an example of plug-in development, refer to the Sun BluePrints™ OnLine article titled "Writing an Authentication Plug-in for a Sun ONE Directory Server."

Considering Purpose and Design

Our purpose is to design a framework for triggers in the Sun Java System Directory Server using its Plug-in API as a foundation. We might call them LDAP triggers. With triggers, we mean an abstraction that allows us to define an action that must be invoked when an event occurs. The action might be simply recording an entry in a change log, sending a notification to an administrator, or changing an attribute of another logically associated entry. Computer scientists like to remark that by means of triggers, a data repository is turned into an active database; that is, the repository's state transactions are not only a consequence of external actions, but also the result of internal events.

Overall a framework for triggers does the following:

- Supports an administrator's manipulation of triggers (creation, deletion, inactivation, and so on.)
- Supports predefined actions (for example `IGNORE`, skip the LDAP operation, `LOG` to log the action, and so on)
- Supports custom actions (for example, the ability to execute user functions loaded from external libraries)
- Allows user interaction based on a SQL-like language

In our example, triggers are administrator's objects (replication agreements are an example of administrator’s objects), and they cannot be instantiated by non-privileged users. To enable this feature, a non-trivial effort is required to manage permissions and rights.

To design a trigger framework, initially consider the following requirements:

1. The administrator, through a client console or command line application, can manipulate triggers through simple commands (create, delete, enable, and so on).
2. The framework asks predefined actions that an administrator can choose from when creating triggers.

3. The framework asks for custom actions, possibly implemented by externally, user-provided libraries.

4. A language for the trigger manipulation, that is, a collection of trigger instructions (such as `CREATE TRIGGER`, `DELETE TRIGGER`, and so on) that simplify user interaction with trigger objects.

After we are certain that we know what the customer wants, we ask: “Do we have the right tool or technologies to realize this framework in Sun Java System Directory Server?” and “Is the solution feasible?”

A UNIX expert would easily figure out which tool is needed to create a small language in C: YACC. It allows you to specify a grammar for your language, that, with a lexical analyzer specification provided by you, greatly simplifies the task of writing a compiler. In fact, the YACC task is to generate a compiler for you; it is the kind of tool often referred to as the compiler of compilers.

We have the parser that parses the trigger instructions, but we need a client command-line-based application to accept administrator commands to be parsed. At first we might be tempted to embed the parser in the client application, but this approach is not a good solution. Instead, it’s better to run the parsed commands directly in the server, embedding them into a plug-in. This approach permits third parties to write their own client applications. The database server follows a similar architectural solution: third parties embed the SQL parser component in the server itself, and allow client applications, written in different languages and for various platforms, to send plain SQL text.

How do you send the commands the administrator enters in the client application to the LDAP server? Let’s put it into context with an extended operation. We could pass the trigger statements to the server-side parser through an extended operation invocation, with the invocation argument being the trigger commands.

In our design, the parser is responsible only for parsing the instructions it turns into statements, like `CREATE TRIGGER`, in a data structure for an internal LDAP operation. That is, if an administrator issues a command such as `DELETE TRIGGER mytrigger`, we program the parser to produce a data structure, for example, `ParserOutput`, which is passed down to a routine that performs the actual data operation.

Intuitively, in a directory server we could record the triggers as LDAP entries. Standard schema, however, does not have an object class suitable for our objects (triggers), therefore we need to extend it.

The object class definition chosen for triggers as it might appear in the `99user.ldif` user schema file is as follows:

```
objectClasses: ( trigger-oid NAME 'trigger' SUP top STRUCTURAL
MUST ( cn $ enabled )

 MAY ( action $ actiondn $ before $ explain $ externallib $ on )
X-ORIGIN 'user defined' )
```

The information we choose to store about triggers is as follows:

TABLE 1 Storing Trigger Information

Item	Description
`cn`	Its name, as typed by the user
`enabled`	Flag that indicates if the trigger is active
`operation`	The trigger run on this LDAP operation (`ADD`, `DELETE`, etc.)
`on`	The DN of the entry for which the trigger is registered
`action`	Action code that specifies which action to perform
`actiondn`	For some actions, it indicates on which DN the action must be performed
`before`	`1` if this is before a trigger; `0` is it is after a trigger
`externallib`	When the action is external, this attribute contains the name of the library to load
`explain`	As the name suggests, it stores the instruction that created this trigger (it is retrievable by using the `explain trigger <`*`trgname`*`>` command

Triggers are stored flatly under a specific branch, `ou=triggers,o=sunblueprints` in the previous code example.

Of course when a data event occurs, for example, deleting an entry, nothing happens unless we override the default LDAP operation. That's why we need a pre-operation (before triggers) and post-operation (after triggers) plug-in, which is merged in a single plug-in. We address this in the next section. The pre- and post-operation plug-in is the most important component of the framework, because it is where the trigger runs.

Now we have all the pieces to start building a solution. First, let's summarize how the various components functionally cooperate and concur to serve user requests. The logical flow is as follows:

- The administrator enters trigger statements at the client application prompt.
- The client application issues an extended operation call, passing as argument the trigger statements verbatim.
- On the server side, the extended operation routine invokes the parser to parse the user commands, which turns them into trigger data structures and executes all the trigger operations.
- The client application receives a response and reports it to the user.
- When an event occurs (an LDAP operation is issued by a user), the pre- and post-operation plug-in searches for triggers registered with the DN of the entry being affected by the data event. If found, it applies the actions associated with the triggers.

In this article, we implement the framework in our code example to provide an advanced example of implementing and using extended operations and directory plug-ins.

Extended Operations

Extended operations are extension mechanisms defined by the LDAP standards (refer to RFC 2251). They were defined to implement new operations, beyond the standard ones (`ABANDON`, `SEARCH`, `MODIFY`, `MODRDN`, `DELETE`, `BIND`, `UNBIND`).

To write your extended operations, use the Sun Java System Directory Server Plug-In API. Developers who are familiar with the Plug-In API typically find it easy to learn how to implement extended operations, because they are just special plug-ins. All the code, makefiles, and utilities you built to simplify your work will probably be reusable.

The only relevant difference with these plug-ins compared to pre- and post- plug-ins is that extended operations are explicitly and directly invoked by clients, while pre- and post- plug-ins act in the middle of LDAP operations. Consequently, the client must have a mechanism to invoke extended operations. The client C SDK or Java SDK, now part of the Sun Java System Directory Server Resource Kit, includes the library calls that allow you to invoke extended operations. Also, there are basic examples that show you the minimum amount of code to write to make a call and get a response.

Clients can discover which extended operations are available on a LDAP Server, by querying the Directory Specific Entry (DSE). Here is the search that retrieves the list of extended operations available on Sun Java System Directory Server, on which we installed the extended operation example:

CODE EXAMPLE 1 Search for List of Extended Operations

```
<nico ldap-triggers>~$ ldapsearch -p 10389 -D "cd=Directory
Manager" -w manager0 -s base -b "" "objectclass=*"
supportedExtension

supportedExtension: 2.16.840.1.113730.3.5.7
supportedExtension: 2.16.840.1.113730.3.5.8
[...cut...]
supportedExtension: 1.3.6.1.4.1.42.2.27.9.6.22
supportedExtension: 4.3.2.1
supportedExtension: 1.3.6.1.4.1.4203.1.11.3
```

The extension "`4.3.2.1`" is our implementation. You might have noticed how its number has a different pattern when compared with the others listed in the output. In fact it is a number invented for the purpose of our example; instead, it should be unique and officially registered, to avoid conflict with other registered services.

On the client side, the job is fairly easy, because you only need to invoke a library call with all its requested parameters, then either process the resulting data or manage an exception, in case of unexpected error. The difficult job is on the server side, in the extended operation routine itself, which you have to write. To implement an extended operation, you need to do the following:

- Write a C function, using the Plug-In API
- Register the function in the server (through a standard mechanism)
- Handle the call when it is invoked by clients

To register your extended operation at runtime, write an initialization routine that gets called by the server at startup, giving you the opportunity to register and set up the data structure for your operations.

The following code example illustrates the initialization routine in `triggers_extop.c`, one of the source files in our example.

CODE EXAMPLE 2 Initialization Routine in `triggers_extop.c`

```
#include <stdio.h>                 /* Standard C headers       */
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <slapi-plugin.h>       /* SunONE DS Plugin-API 5.2  */
#include "slapi_utils.h"        /* Our logging utilities,
log_info_conn, log_error ... */
#include "triggers_impl.h"      /* Triggers data structures */
#include "triggers_extop.h"

char *trgbase;
...

int triggers_extop_init(Slapi_PBlock * pb)
{
...

    trgbase = slapi_ch_strdup(argv[0]);

    /* Extended operation plug-ins may handle a range of OIDs. */
    oid_list = (char **)slapi_ch_malloc((argc) * sizeof(char *));
    for (i = 1; i < argc; ++i) {
        oid_list[i] = slapi_ch_strdup(argv[i]);
        log_info_conn(pb,
            "triggers_extop_init",
            "Argv[%i]= %s.",
            oid_list[i]
        );
    }
    oid_list[argc - 1] = NULL;

   rc |= slapi_pblock_set(          /* Extended op. handler    */
        pb,
        SLAPI_PLUGIN_EXT_OP_FN,
        (void *) trg_service_fn
    );
   rc |= slapi_pblock_set(          /* List of OIDs handled    */
        pb,
        SLAPI_PLUGIN_EXT_OP_OIDLIST,
        oid_list
    );
```

CODE EXAMPLE 2 Initialization Routine in `triggers_extop.c` *(Continued)*

```
#include <stdio.h>                  /* Standard C headers       */

...
}
```

Note – To save space in this article, we omit irrelevant code such as the code for log and error management.

The routines `log_info`, `log_error`, `log_warning`, and so on in `slapi_utils.c` source are shorter forms of the widely used `slapi_log_info_ex`, `slapi_log_warning_ex`, and `slapi_log_error_ex` functions of the Plug-In API. If you use the official log routines, they have a much longer parameter list and you need an extra call to retrieve some of them. With `slapi_utils.c` functions, you write less lines and the code is a bit more readable.

The initialization code is divided into two parts:

- Get information: plug-in identity and arguments
- Set information: version, extended operation routine address, and list of Object IDentifiers (OIDs) handled by the routine

The code is similar to the code example in `testextendedop.c`, located in the plug-in directory of any Sun Java System Directory Server installation. The one notable difference is that this plug-in requests a plug-in ID, which is necessary if you need to invoke internal server operations (internal operations are LDAP operations not initiated by user requests). This plug-in code uses server internal operations, therefore a plug-in ID is mandatory.

The function `trg_service_fn` is our extended operation routine. The signature is simple because the server passes all the information through a `SLAPI` parameter block (which is the only routine argument), and you use `slapi_pblock_get` to obtain from it the data you need and expect. The plug-in requires an argument list where the first element is the base DN for its operations (for example, `ou=triggers,o=sunblueprints`), and the rest are OIDs of extended operations implemented in plug-ins. We implement only one extended operation, even though the code supposes many.

To register your extended operation plug-in, you need an LDAP Data Interchange Format (LDIF) file, which defines an add operation under `cn=plugins, cn=config`. If you prefer, perform the add directly from the command line, without storing instructions in a LDIF file. The content of the LDIF file is as follows:

CODE EXAMPLE 3 LDIF File Contents

```
dn: cn=Triggers ExtendedOp,cn=plugins,cn=config
changetype: add
cn: TriggersExtendedOp
objectclass: top
objectclass: nsSlapdPlugin
objectclass: extensibleObject
nsslapd-pluginpath: <server_root>/lib/triggers_extop.so.1.0
nsslapd-pluginInitfunc: triggers_extop_init
nsslapd-pluginType: extendedop
nsslapd-plugin-depends-on-type: database
nsslapd-pluginEnabled: on
nsslapd-pluginId: triggers_extop
nsslapd-pluginVersion: 1.0
nsslapd-pluginVendor: Sun Microsystems Italy
nsslapd-pluginDescription: Sun Blueprints Triggers Extended
Operation Plugin
nsslapd-pluginArg0: ou=triggers,o=SunBlueprints
nsslapd-pluginArg1: 4.3.2.1
```

Among other things, we define plug-in name, plug-in ID, where the library containing the functions is on the file system, what is the initialization function, and what is the plug-in type. The last two attributes make the plug-in argument list. As mentioned previously, OID `4.3.2.1` is a unique unregistered OID for our extended operation. If you want to change the argument list later, after the plug-in deployment, you can use the Administrator's Console using the “Configuration” tab. Refer to the Administrator's Console documentation.

Before illustrating what our extended operation routine does, we want to focus on the purpose and design of our code example.

Implementing a Solution

After determining the purpose and considering design issues, implement your solution. As mentioned earlier, the main components of a solution are as follows:

- Client application
- Parser
- Extended operation
- Pre- and post-operation plug-ins

For our example, we write the client application in Java, using the Java™ Netscape API for the Sun Java System Directory Server. The following is a code excerpt for our client example:

CODE EXAMPLE 4 Client Application Sample

```
...
    try
    {
        ld = new LDAPConnection();
        // some basic setup and info
        miscellaneus(ld);

        // connect to the server
        System.out.println("Connecting to \'" + host + ":" + port
+"\'");
        ld.connect( host, port );

        // binds as with admin credentials
        // Remember to specify LDAPv3,
        // otherwise you will not be able
        // to use extended operatins
        System.out.println("Binding as \'" + binddn +"\'");
        ld.bind(3, binddn, bindpw);

        // in loop for user input
        String  data = getUserInput();
        byte[] databytes = data.getBytes("UTF8");

        // prepare the extended operation object
        LDAPExtendedOperation extop = new
            LDAPExtendedOperation(extopOID, databytes);

        // invoke the server extended operation
```

CODE EXAMPLE 4 Client Application Sample *(Continued)*

```
        System.out.println("Calling extended operation " +
extopOID);
        LDAPExtendedOperation extres =
ld.extendedOperation(extop);

        // get results
        String id = extres.getID();
        String response = new String(extres.getValue(), "UTF8");

        System.out.println("Returned OID: " + id);
        System.out.println("Returned Data: " + response);

        // disconnect
        ld.disconnect();
    }
...
```

When started, the program shows the `TRIGGERS>>` prompt and waits for your statements. To execute the program, use the `makefile run` target, which sets up the `classpath` for you and runs the main Java class:

CODE EXAMPLE 5 Executing the Program

```
<nico client>~$ make run
/usr/java/bin/java -cp /opt/sunone/ldapjdk41/packages/
ldapjdk.jar:$CLASSPATH TriggersClient

Proprietary protocol: 3.0
LDAP SDK: 4.1
Logging LDAP activity on file +error.log
Connecting to 'localhost:10389'
Binding as "cd=Trigger Manager"
Triggers Client
Insert your commands, a line with '.' terminates input

triggers>>
```

Note – The application manages triggers in Sun Java System Directory Server using the account `cn=Trigger Manager`. Create this in the server, and grant it all permissions for the `ou=triggers, o=sunblueprints` branch.

To terminate a command list, type a line with just a ".", as shown in the following example:

CODE EXAMPLE 6 Terminating a Command List

```
...
TRIGGERS>>  create or replace trigger BeHappyWithSunBlueprints on
'o=sunblueprints' before ldap_add action ignore;
disable trigger myspecialtrg;
! cat /etc/hosts \;
.
...
```

The example shows a special feature of our language: the ! escape feature. Issuing a command escape by using a ! results in the execution of a UNIX command on the server. In this case, you would see the host database of the LDAP server machine on your client terminal.

After the ".", the program runs the extended operation, turns the resulted data into UTF8 representation, and displays it to the user on the terminal.

The UNIX command is just one possibility of our language. The grammar of the trigger language is partially shown in the following example:

CODE EXAMPLE 7 Trigger Language Grammar

```
cmdlist:
    cmd ';'
    |
    cmdlist cmd ';'
    ;

cmd:
    trgcmd
    |
    cntlcmd
    |
    unixcmd
    ;

trgcmd:
    CREATE OR REPLACE TRIGGER STRING ON what BEFORE operation
ACTION action
       {
           CURRENT_STATEMENT.cmdcode = TRIGGER_CREATE;
           strcpy(CURRENT_STATEMENT.name, $5);
           strcpy(CURRENT_STATEMENT.what, $7);
```

CODE EXAMPLE 7 Trigger Language Grammar *(Continued)*

```
            strcpy(CURRENT_STATEMENT.operation, $9);
            CURRENT_STATEMENT.before = 1;
            sprintf($$, "%s %s %s %s %s %s %s %s %s %s",
            $1, $2, $3, $4, $5, $6, $7, $9, $10, $11);
        }
    |
    CREATE OR REPLACE TRIGGER STRING ON what AFTER operation ACTION
action
        {
            CURRENT_STATEMENT.cmdcode = TRIGGER_CREATE;
            strcpy(CURRENT_STATEMENT.name, $5);
            strcpy(CURRENT_STATEMENT.what, $7);
            strcpy(CURRENT_STATEMENT.operation, $9);
            CURRENT_STATEMENT.before = 0;
            sprintf($$, "%s %s %s %s %s %s %s %s %s %s",
            $1, $2, $3, $4, $5, $6, $7, $9, $10, $11);
        }
    |
    ENABLE TRIGGER STRING
    |
    DISABLE TRIGGER STRING
    |
    DELETE TRIGGER STRING
    |
    LIST ALL TRIGGERS
    |
    EXPLAIN TRIGGER STRING
    ;


what:
    '"' QSTRING '"'
    |
    ANY ENTRY
    ;


operation:
    LDAP_ADD
    |
    LDAP_DELETE
    |
    LDAP_MODIFY
    |
    LDAP_MODRDN
    |
```

CODE EXAMPLE 7 Trigger Language Grammar *(Continued)*

```
    LDAP_BIND
    |
    LDAP_UNBIND
    |
    LDAP_ABANDON
    ;


action:
    DELETE ’”’ QSTRING ’”’
    |
    DELETE ’”’ QSTRING ’”’ ON ERROR CONTINUE
    |
    LOG
    |
    IGNORE
    |
    EXECUTE EXTERNAL LIB STRING
    |
    EXECUTE EXTERNAL LIB STRING ON ERROR CONTINUE
    ;


cntlcmd:
    SET STRING ’=’ STRING
    |
    SET STRING ’=’ NUMERIC
    |
    GET STRING
    ;

unixcmd:
    ’!’ STRING
    |
    ’!’ STRING cmdargs
    ;

cmdargs:
    STRING
    |
    cmdargs STRING
    ;

%%
```

Note – For clarity, the grammar in this code example does not show all actions.

Our language is a list of statements separated by semicolons “`;`”. Then, each command can be one of the following:

- A standard trigger command (for example, `CREATE OR REPLACE TRIGGER`)
- A control command (for example, to set the environment with `SET var=name`)
- A UNIX command (a command with “`!`” escape symbol)

The rest of the grammar defines commands and other syntactical components. If you are interested in complete details, the grammar is in the `triggers.y` file.

The `LEXer` specification file tells LEX how to extract tokens from the source text; the specification file is `triggers.l`.

Note – Unfortunately YACC and LEX are not designed for a concurrent environment.

Another limitation is that LEX expects its input to come from `yyin` global variable, which is a `FILE * object`; however, our input is a string passed by the client application, and we are forced to create a temporary file, write in the text, and assign the handler to the `yyin` global variable. A way to work around this is to define your own `input()` routine, which LEX uses in place of its default. However, this extension mechanism greatly depends on the version of LEX. If you want to remove any nonre-entrant problem from your code, use one of the latest versions of LEX, which have been redesigned.

Note – The code in `{}` parenthesis represents the C code that we want the parser to execute after a command or statement is read.

The extended operation is responsible for the following tasks:

- Retrieving its argument
- Passing it to the parser
- Getting the parser output
- Calling a coded function for any of the parsed statements

The following code shows how `trg_service_fn`, the extended operation routine, works:

CODE EXAMPLE 8 How `trg_service_fn` Works

```
int trg_service_fn(Slapi_PBlock *pb)
{

    char          * oid = NULL;                                 /*
Client request OID       */
    struct berval * client_bval = NULL;    /* Value from client
*/
    char          * result = NULL;     /* Result to send to client
*/
    struct berval * result_bval = NULL;/* Encoded result
*/
    int             rc = 0;
    int             i;
    int             len;
    ParsedStatements stmts;/* Result of the parsing */
    char msgbuf[MAX_MSGBUF];

      /* log */
       log_info_conn(pb,
    "trg_service_fn",
    "Entering pb=%p", pb);

    /* Get the OID and the value included in the request.
     * The client_bval will contain the sequence of statements
     * submitted by the client
      */
     log_info_conn(pb,
    "trg_service_fn",
    "Getting ext opt request data...");
    rc |= slapi_pblock_get(pb, SLAPI_EXT_OP_REQ_OID,   &oid );
    rc |= slapi_pblock_get(pb, SLAPI_EXT_OP_REQ_VALUE,
&client_bval);
    if (rc != 0)
   {
         snprintf(msgbuf,
    MAX_MSGBUF,
    "Unable to get client OID/Value data: %d",
    rc);
         goto TRIGGERS_ERROR;
    }

    log_info_conn(
```

CODE EXAMPLE 8 How `trg_service_fn` Works *(Continued)*

```
                              pb,
       "trg_service_fn",
      "Request with OID: %s  Value from client: %s\n",
      oid, client_bval->bv_val);


/* Parsing the client statements
 * We invoke a separate routine to parse the client
 * statements.
 * The statements are stored in the stms structure  */
stmts.statements = 0;
rc = trg_parse_client_req(pb, client_bval, &stmts);
if(rc != OK)
{
                           snprintf(msgbuf,
MAX_MSGBUF,
"trg_parse_client_req failed(%d)",
rc);
  goto TRIGGERS_ERROR;
}


/* Is at this point, parsing was successful.
 * Now we invoke the routine that creates triggers
 * as LDAP objects
 */
rc = trg_apply_statements(pb, &stmts, &result_bval);
if(rc != OK)
{
                          snprintf(msgbuf,
MAX_MSGBUF,
"trg_apply_statements failed(%d)",
rc);
   goto TRIGGERS_ERROR;
}

/* Calculate the amount of text data
 * to be returned to the client */
for(i = 0, len = 0; i < stmts.statements; ++i)
{
len += sprintf(msgbuf, "Statement #%d\n", i) - 1;

len += strlen("Error code: ");
len += sprintf(msgbuf, "%d\n", i) - 1;

len += strlen("Error msg: ");
```

CODE EXAMPLE 8 How `trg_service_fn` Works *(Continued)*

```
    len += strlen(stmts.opres[i].errmsg);
    len += 1; /* \n */

    len += strlen(stmts.opres[i].out);
    len += 1; /* \n */
    len += 1; /* \n */
    }

    /*
    * Set the value to return to the client, depending on what your
     * plug-in function does. Here, we return the value sent by the
    * client, prefixed with the string "Value from client: ".    */
    result  = (char *)slapi_ch_malloc(len + 1);
    for(i = 0, result[0] = '\0'; i < stmts.statements; ++i)
    {
    sprintf(msgbuf, "Statement #%d\n", i);
    strcat(result, "Error code: ");

    sprintf(msgbuf, "%d\n", i);
    strcat(result, msgbuf);

    strcat(result, "Error msg: ");
    strcat(result, stmts.opres[i].errmsg);
    strcat(result, "\n");

    strcat(result, stmts.opres[i].out);
    strcat(result, "\n");
    }


    log_info_conn(pb,
    "trg_service_fn",
    "Client statements executed");

    result_bval = (struct berval *)slapi_ch_malloc(
        sizeof(struct berval));

    result_bval->bv_val = result;
    result_bval->bv_len = strlen(result_bval->bv_val) + 1;

   /* Prepare the PBlock to return and OID and value to the client.
    * Here, we demonstrate that the plug-in may return a different
     * OID than the one sent by the client. You may, for example,
    * use the different OID to indicate something to the client.  */
    rc |= slapi_pblock_set(pb, SLAPI_EXT_OP_RET_OID,   "4.3.2.1");
```

CODE EXAMPLE 8 How `trg_service_fn` Works *(Continued)*

```
    rc = slapi_pblock_set(pb, SLAPI_EXT_OP_RET_VALUE,
result_bval);
    if (rc != 0)
   {
        snprintf(msgbuf,
    MAX_MSGBUF,
    "Unable to set extended operation result value: %p",
    result_bval);
      goto TRIGGERS_ERROR;
    }

    /* Log data sent to the client */
    log_info_conn(
        pb,
        "trg_service_fn",
        "OID sent to client: %s  \nValue sent to client:\n%s",
        "", result);

    /* Send the result back to the client */
    slapi_send_ldap_result(
        pb,
        LDAP_SUCCESS,
        NULL,
        result,
        0,
        NULL);

    /* Rree memory allocated for return berval  */
    if(result) slapi_ch_free_string(&result);
    if(result_bval) slapi_ch_free((void **)&result_bval);

    goto TRIGGERS_OK;

TRIGGERS_ERROR:

    log_error_conn(pb,
    0,
    "trg_service_fn",
    "Routing failed ",
    msgbuf);

    /* Sends back an error-result */
    slapi_send_ldap_result(
    pb,
    LDAP_OPERATIONS_ERROR,
    NULL,
```

CODE EXAMPLE 8 How `trg_service_fn` Works *(Continued)*

```
    msgbuf,
    0,
    NULL);

    /* Free memory allocated for return berval  */
    if(result) slapi_ch_free_string(&result);
    if(result_bval) slapi_ch_free((void **)&result_bval);

   /* Tell the server that the operation completed */
    return (SLAPI_PLUGIN_EXTENDED_SENT_RESULT);

TRIGGERS_OK:

    /* Tell the server we've sent the result */
    return (SLAPI_PLUGIN_EXTENDED_SENT_RESULT); /* ok! */
}
```

This example is a lot of code, however, apart from the logging and error management code, the logical flow is the following:

- Take the extended operation value – This value contains the user statements, as typed on the client side).
- Prepare and invoke the parser through the `trg_parse_client_req` routine – This routine parses the user commands, and for any command found, fills a structure in the `ParsedStatements` array passed from `trg_service_fn`.
- Call the `routine trg_apply_statements` to apply statements – We have the binary representation of the user commands in a `ParsedStatements` structure, then invoke the `routine trg_apply_statements` to update the directory.
- Prepare the result data for the client – The data has the form of sequential file with records of the following form:

CODE EXAMPLE 9 Sequential File Records Sample

```
Statement: 1
Error code: 0
Error msg: “OK!”
<some output>

Statement: 2
Error code: 1
Error message: Trigger already exists
<some output>
...
```

- Send the results back to the client.
- Free resources.

Notice how we reply back to the client that invoked the extended operation. This reply is in fact a three-part operation:

1. Set the `PBlock` using the `SLAPI_EXT_OP_RET_OID` and `SLAPI_EXT_OP_RET_VALUE` (we could return an extend operation OID different from the client.)
2. Invoke the usual `slapi_send_ldap_result()`, which sends the result to the client.
3. Tell the server that we are done, returning `SLAPI_PLUGIN_EXTENDED_SENT_RESULT`.

The first routine invoked is `trg_parse_client_req` and contains the code that instantiates and runs the parser through the `yyparse()` routine, which is generated by YACC as part of the project's building.

The code is a bit tricky because of the nonre-entrant code generated by YACC: we are forced to use a lock (`mutex`) and generate a temporary file. The file pointer is then passed via the global variable `yyin`. The goal of `trg_parse_client_req` is to either fill a `ParsedStatements` structure with statements sent by the client or to return an appropriate error if the parser fails. For production, consider embedding a re-entrant parser in your extended operation plug-in; this approach is more suitable for a concurrent environment.

The other helper function is the `trg_apply_statements` routine. By itself, `trg_apply_statements` does not contain much logic, it simply runs a list of statement structures and switches on the value of the operation code associated with the statement. Based on this code, a routine is called to do the last job, which is recording the trigger in LDAP.

The following shows an example of `trg_execute_create`, which creates a trigger in the server:

CODE EXAMPLE 10 Creating a Trigger in the Server

```
static int trg_execute_create(Slapi_PBlock *pb, int index,
ParsedStatements *stmts)
{
    int rc;
...

    char *base = "ou=triggers,o=sunblueprints";
    char msgbuf[MAX_MSGBUF];
    Slapi_DN *sdn = NULL;
```

CODE EXAMPLE 10 Creating a Trigger in the Server *(Continued)*

```
static int trg_execute_create(Slapi_PBlock *pb, int index,
ParsedStatements *stmts)
    Slapi_Entry *entry = NULL;
    Slapi_PBlock *pbop = NULL;
    char *entrydn = NULL;
    char *entryldif = NULL;
    int len;

    if(!stmts || index < 0 )
    return ERR;

    name = stmts->output[index].name;
    on = stmts->output[index].what;
    before = stmts->output[index].before;
    action = stmts->output[index].action;
    action_dn = stmts->output[index].action_dn;
    external_lib = stmts->output[index].external_lib;

    if(!name || !on )return ERR;

    /* Make a SDN of the string dn */
    sdn = slapi_sdn_new_dn_byval(base);
    if(sdn == NULL)
    {
    snprintf(msgbuf,
           MAX_MSGBUF,
           "Unable to create a SDN from the string %s",
           base);
    goto CREATE_ERR;
    }

...
    /* prepare the entry to be added
     * using data provided by user */
    entrydn = slapi_ch_malloc(
        + strlen("cn=")
        + strlen(name)
        + 1
        + strlen(base)
        + 1);
    sprintf(entrydn, "cn=%s,%s", name, base);
    entry = slapi_entry_alloc();
    slapi_entry_set_dn(entry, strdup(entrydn));

    /* Set entry attributes. Note that slapi_entry_add_string
     * does not consumes the string passed as parameter,
```

CODE EXAMPLE 10 Creating a Trigger in the Server *(Continued)*

```
static int trg_execute_create(Slapi_PBlock *pb, int index,
ParsedStatements *stmts)
     * hence we do not need a slapi_ch_strdup call */
    slapi_entry_add_string(entry, "objectclass", "trigger");
    slapi_entry_add_string(entry, "cn", name);
    slapi_entry_add_string(entry, "on", on);
    slapi_entry_add_string(entry, "before", before ? "1" : "0");
    slapi_entry_add_string(entry, "enabled", "true");

    snprintf(msgbuf, MAX_MSGBUF, "%d", action);
    slapi_entry_add_string(entry, "action", msgbuf);
    slapi_entry_add_string(entry, "actiondn", action_dn);

    /* Preparing PBlock for internal add
     * The plugin_id parameter was obtained
     * int the _init function */
    pbop = slapi_pblock_new();
    slapi_add_entry_internal_set_pb(
    pbop,
    entry,
    NULL,
    plugin_id,
    SLAPI_OP_FLAG_NEVER_CHAIN);

    /* capture the entry's LDIF rapresentation
     * before the internal add, which consumes the entry  */
    entryldif = slapi_entry2str(entry, &len);

    /* Perform the internal add: which consists
     * of calling slapi_add_internal_pb and getting
     * the operation result from the slapi_pblock_get */
    log_info_conn(
    pb,
    "trg_execute_create",
    "Adding the entry ..." );
    rc = slapi_add_internal_pb(pbop);
    slapi_pblock_get(pbop, SLAPI_PLUGIN_INTOP_RESULT, &rc);
    if(rc)
    {
    snprintf(msgbuf,
        MAX_MSGBUF,
        "slapi_pblock_get reported an error(%d) for the previous
internal op",
        rc);
    goto CREATE_ERR;
    }
```

CODE EXAMPLE 10 Creating a Trigger in the Server *(Continued)*

```
static int trg_execute_create(Slapi_PBlock *pb, int index,
ParsedStatements *stmts)

    log_info_conn(
    pb,
    "trg_execute_create",
    "Created entry:\n %s",
    entryldif);

    /* Setting return data */
    stmts->opres[index].errcode = 0;
    snprintf( stmts->opres[index].errmsg,
    MAX_ERR_MSG,
    "Trigger %s successfully added",
    stmts->output[index].name) ;
    stmts->opres[index].out[0] = '\0';
    /* Freeing some allocated memory
     * and jumping to the ok label */
    if(sdn) slapi_sdn_free(&sdn);
    if(pbop) slapi_pblock_destroy(pbop);

    goto CREATE_OK;
...
```

As we said previously, the trigger is executed before/after a standard LDAP operation, and it is the responsibility of our pre- and post-operation plug-in to understand whether a trigger must be activated for an operation on an LDAP entry.

To fulfill this goal, pre- and post-operation plug-ins during initialization need to register all pre- and post-operation functions.

CODE EXAMPLE 11 Registering Extended Operation Functions

```
...
    /* Register postoperation routines
     * We override all LDAP Operations */
    rc = slapi_pblock_set(pb, SLAPI_PLUGIN_PRE_ADD_FN,
    (void *)triggers_pre_add_fn);

    rc |= slapi_pblock_set(pb, SLAPI_PLUGIN_PRE_MODIFY_FN,
    (void *)triggers_pre_modify_fn);

    rc |= slapi_pblock_set(pb, SLAPI_PLUGIN_PRE_DELETE_FN,
    (void *)triggers_pre_delete_fn);
...
```

CODE EXAMPLE 11 Registering Extended Operation Functions *(Continued)*

```
...
    rc |= slapi_pblock_set(pb, SLAPI_PLUGIN_POST_ABANDON_FN,
    (void *)triggers_post_abandon_fn);

    rc |= slapi_pblock_set(pb, SLAPI_PLUGIN_POST_BIND_FN,
    (void *)triggers_post_bind_fn);

    rc |= slapi_pblock_set(pb, SLAPI_PLUGIN_POST_UNBIND_FN,
    (void *)triggers_post_unbind_fn);
...
```

The registrations of pre- and post-operation functions are placed in two different routines: `triggers_postop_init` and `triggers_pre_init`. This action is necessary so that we can register two plug-ins: a pre-operation plug-in and a post-operation plug-in, even though the code is in a single source file and the shared object is the same.

When a pre-operation function like `triggers_pre_add_fn` gets called because of a user LDAP add operation, it calls a helper function called `find_trigger_by_targetdn()`, which searches for a trigger associated with the DN of the entry that the user is trying to add. If there is such a trigger, the routine executes the action defined by the trigger. The following is an excerpt of the code.

CODE EXAMPLE 12 Using Helper Functions

```
...
    if(!find_trigger_by_targetdn(pb, &cb_data, dn,
BEFORE_TRIGGER)
        && cb_data.nentries > 0)
    {
    /* STEP 2.1. Take the action associated
     * to this trigger */
    switch(cb_data.tr.action)
    {
        case ACTION_DELETE_DN:
            /* Deleate some entry logically associated
             * with entry being removed */

            /* LEFT TO BE DONE AS AN EXERCISE */
            break;

        case ACTION_LOG:
            /* Get added entry from the operational block */
            rc = slapi_pblock_get(pb, SLAPI_ADD_ENTRY, &entry);
            if(rc != 0)
```

CODE EXAMPLE 12 Using Helper Functions *(Continued)*

```
...
            {
            sprintf(msgbuf,
                "slapi_pblock_get(SLAPI_ADD_ENTRY,..)=%d",
                rc);
            err = rc;
            goto PRE_ADD_ERR;
            }

            /* Get LDIF representation of the added entry */
            ldif = slapi_entry2str(entry, &len);
            log_info_conn(pb,
                "TRIGGERED ACTION",
                "THE FOLLOWING ENTRY HAS BEEN ADDED:\n------ NEW
ENTRY -----\n%s\n",
                ldif);
            break;

        case ACTION_IGNORE:

            sprintf(msgbuf, "Trigger %s forbids the creation of
entry %s",
                cb_data.tr.name,
                dn);

            slapi_send_ldap_result(
                pb,
                LDAP_OPERATIONS_ERROR,
                NULL,
                msgbuf,
                0,
                NULL );
            return ERR;
            break;

        case ACTION_EXTERNAL:
            /* Load an external library, and execute
             * a named function inside it */

            /* LEFT AS AN EXERCISE */
            break;

        default:
            log_warning_conn(pb,
                0,
                "triggers_pre_add_fn",
```

CODE EXAMPLE 12 Using Helper Functions *(Continued)*

```
...
                "Unrecognized action",
                "No action taken");
            break;

    };
    }
...
```

To search the trigger under `ou=triggers,o=sunblueprints`, we use an internal search operation (not shown, but in `find_trigger_by_targetdn`).

A few actions are implemented in our example. In the panel you see, for example, the `LOG` action, which consists of writing the entry to be added in LDIF format in the error log file. Another more interesting action implemented is `IGNORE`, which allows you to skip the operation. What this means is that the entry is not added, and the user receives an error message such as "`Trigger stopItTrigger forbids you to add this entry`."

More advanced action can now be easily added; it is just a matter of adding code in this routine. The framework does the rest of the work for you.

Testing the Code

Now let's test the code and play with some commands. Before testing the code, remember to enable the plug-in `LOG` level in the LDAP server, from the administrator's console or from the command line as follows:

CODE EXAMPLE 13 Enabling the Plug-In Log Level

```
<sunone slapd-sunblueprints>~$ ldapmodify -p 10389 -D
"cn=Directory Manager" -w manager0
dn: cn=config
changetype: modify
replace: nsslapd-infolog-area
nsslapd-infolog-area: 65536
^D
```

Let's try a UNIX command. The language is defined to accept UNIX escape commands such as the following:

```
! pwd \;
! man cat \;
```

Many UNIX programs support this useful feature (for example, `mail`, `vi`).

The following example executes two commands. Notice that the ! escape symbol is expected at beginning of the line and a terminating '`\;`' sequence is required. Refer to the `triggers.l` file and the `LEXer` specification file for the lexical details of the language.

CODE EXAMPLE 14 Executing Commands

```
triggers>> !uname -a \;
! man pwd \;
.
Calling extended operation 4.3.2.1
------------- Your input ----------------
!uname -a \;
! man pwd \;

.
-----------------------------------------

------------- Server output ----------------
Statement #0
Error code: 0
Error msg: Unix shell command successfully executed
Linux alfa 2.4.18-3 #1 Thu Apr 18 07:37:53 EDT 2002 i686 unknown

Statement #1
Error code: 0
Error msg: Unix shell command successfully executed
PWD(1)                                                  PWD(1)



NOME
       pwd - stampa il nome della directory di lavoro corrente

SINTASSI
       pwd
       pwd {--help,--version}

DESCRIZIONE
```

CODE EXAMPLE 14 Executing Commands *(Continued)*

```
       Questa  documentazione  non  ?  mantenuta da lungo tempo e
       potrebbe essere inaccurata o incompleta. La documentazione
       in Texinfo ? ora la fonte autorevole.

       Questa pagina di manuale documenta la versione GNU di pwd.
       pwd stampa il nome della directory  corrente  risolvendolo
       completamente.  Cio?,  tutte le componenti del nome stam-
       pato saranno nomi di directory reali --  nessuna  sar?  un
       link simbolico.

       Si noti che molte shell Unix forniscono un proprio comando
       pwd interno con funzionalit? simili cosicch? il  semplice,
       interattivo  comando  pwd  di  solito eseguito sar? quello
       della shell e non questo.

   OPZIONI
       --help Mostra nello standard output un  messaggio  d'aiuto
              ed esce con successo.

       --version
              Mostra  nello  standard  output  informazioni sulla
              versione ed esce con successo.


FSF                   GNU Shell Utilities                PWD(1)

------------------------------------------
Triggers Client
Insert your commands, a line with '.' terminates input
```

We executed the commands `uname -a` and `man pwd`. Both are executed on the server, where Sun Java System Directory Server and the extended operation run. You could use this feature for a minimal remote administration if you like, but our goal was just to show an advanced use of the extended operation. Consider also that this kind of feature could introduce serious security concerns, especially if the remote server runs with `root` account privileges.

Now let's review the log error file to determine what happened.

CODE EXAMPLE 15 Reviewing the Log File

```
[20/Oct/2003:01:47:58 +0200] - INFORMATION - (PID=3018,ThID=13325)
trg_service_fn  - conn=2 op=1 msgId=1 -  Entering pb=0x8354d60
[20/Oct/2003:01:47:58 +0200] - INFORMATION - (PID=3018,ThID=13325)
trg_service_fn  - conn=2 op=1 msgId=1 -  Request with OID: 4.3.2.1
Value from client: !uname -a \;
! man pwd \;
.
[20/Oct/2003:01:47:58 +0200] - INFORMATION - (PID=3018,ThID=13325)
trg_parse_client_req  - conn=2 op=1 msgId=1 -  Entering with
params(0x8354d60,0x8118348,0x42b796fc)
[20/Oct/2003:01:47:58 +0200] - INFORMATION - (PID=3018,ThID=13325)
trg_parse_client_req  - conn=2 op=1 msgId=1 -  Creating the
temporary file for parsing
[20/Oct/2003:01:47:58 +0200] - INFORMATION - (PID=3018,ThID=13325)
trg_parse_client_req  - conn=2 op=1 msgId=1 -  Opening the
temporary file for parsing
[20/Oct/2003:01:47:58 +0200] - INFORMATION - (PID=3018,ThID=13325)
trg_parse_client_req  - conn=2 op=1 msgId=1 -  <<!uname -a \;
! man pwd \;

.>>
[20/Oct/2003:01:47:58 +0200] - INFORMATION - (PID=3018,ThID=13325)
trg_parse_client_req  - conn=2 op=1 msgId=1 -  Now parsing...
[..cut...]
[20/Oct/2003:01:47:58 +0200] - INFORMATION - (PID=3018,ThID=13325)
trg_apply_statements:  - conn=2 op=1 msgId=1 -
trg_apply_statements(0x8354d60,0x42b796fc, 0x43d1191c)
[...cut..]
[20/Oct/2003:01:47:58 +0200] - INFORMATION - (PID=3018,ThID=13325)
trg_execute_unixcmd  - conn=2 op=1 msgId=1 -  Entering (index=0)
[20/Oct/2003:01:47:58 +0200] - INFORMATION - (PID=3018,ThID=13325)
trg_execute_unixcmd  - conn=2 op=1 msgId=1 -  Returning
[20/Oct/2003:01:47:58 +0200] - INFORMATION - (PID=3018,ThID=13325)
trg_execute_unixcmd  - conn=2 op=1 msgId=1 -  Entering (index=1)
[20/Oct/2003:01:47:59 +0200] - INFORMATION - (PID=3018,ThID=13325)
trg_execute_unixcmd  - conn=2 op=1 msgId=1 -  Returning
[20/Oct/2003:01:47:59 +0200] - INFORMATION - (PID=3018,ThID=13325)
trg_apply_statements:  - conn=2 op=1 msgId=1 -  Returning 0
[20/Oct/2003:01:47:59 +0200] - INFORMATION - (PID=3018,ThID=13325)
trg_service_fn  - conn=2 op=1 msgId=1 -  Client statements executed
[20/Oct/2003:01:47:59 +0200] - INFORMATION - (PID=3018,ThID=13325)
trg_service_fn  - conn=2 op=1 msgId=1 -  OID sent to client:
Value sent to client:
```

CODE EXAMPLE 15 Reviewing the Log File *(Continued)*

```
[20/Oct/2003:01:47:58 +0200] - INFORMATION - (PID=3018,ThID=13325)
trg_service_fn  - conn=2 op=1 msgId=1 -  Entering pb=0x8354d60
Statement #0
Error code: 0
Error msg: Unix shell command successfully executed
Linux alfa 2.4.18-3 #1 Thu Apr 18 07:37:53 EDT 2002 i686 unknown

Statement #1
Error code: 0
Error msg: Unix shell command successfully executed
PWD(1)                                                    PWD(1)
...
```

The code traces heavily, and we can see the full stack of calls:

- `trg_service_fn` accepts the extended operation invocation and calls
- `trg_parse_client_req` to parse the client command list, then calls
- `trg_apply_statements`, which in turn, for every user command, issues a call to an execution routine, in this case
- `trg_execute_unixcmd`, which issues `popen` library call to execute the UNIX command and returns the output to
- `trg_service_fn`, which collects all the output from the execution functions in the extended operation response and
- `trg_service_fn` returns the data to the client

This patterns repeats for any command issued by a user at the client application prompt; what changes is the actual `trg_execute_<`*`command`*`>` called.

Now, let's create our first trigger.

CODE EXAMPLE 16 Creating a Trigger

```
TRIGGERS>> create or replace trigger
SunBlueprintsBigTrigger on 'ou=testadd,o=sunblueprints'
before ldap_add action ignore;
.
------------- Server output ----------------
Statement #0
Error code: 0
Error msg: Trigger SunBlueprintsBigTrigger successfully added
-----------------------------------------
TRIGGERS>> list all triggers;
.
------------- Server output ----------------
```

CODE EXAMPLE 16 Creating a Trigger *(Continued)*

```
TRIGGERS>> create or replace trigger
SunBlueprintsBigTrigger on 'ou=testadd,o=sunblueprints'
before ldap_add action ignore;
Statement #0
Error code: 0
Error msg: 1 triggers found
SunBlueprintsBigTrigger
-----------------------------------------
```

The example defines a trigger called `SunBlueprintsBigTrigger` on the directory entry `o=testadd, ou=SunBlueprints` to be triggered before the entry is added to the directory information tree (DIT). As the action, we specify `IGNORE`, which means do nothing or skip the operation.

The `list all triggers` instruction shows you all the available triggers under `ou=triggers,o=sunblueprints`, which is where we store them. As a proof of this, perform the following LDAP search:

CODE EXAMPLE 17 Performing an LDAP Search

```
<sunone slapd-sunblueprints>~$ ldapsearch -p 10389 -D "cd=Trigger
Manager" -w manager0  -b "ou=triggers,o=sunblueprints"
objectclass=trigger"
dn: cn=SunBlueprintsBigTrigger,ou=triggers,o=sunblueprints
objectClass: trigger
objectClass: top
cn: SunBlueprintsBigTrigger
on: ou=testadd,o=sunblueprints
before: 1
enabled: true
explain: create or replace trigger SunBlueprintsBigTrigger on
ou=testadd,ou=sun
 blueprints add action ignore
action: 2
actiondn:
```

The search reports exactly one entry, our trigger. Note the explanation attribute, which stores the whole create instruction, as typed by the administrator. Now create the entry, and see what happens.

CODE EXAMPLE 18 Creating an Entry

```
<sunone slapd-sunblueprints>~$ ldapmodify -p 10389 -D "cd=Trigger
Manager" -w manager0
dn: ou=testadd, o=sunblueprints
changetype: add
objectclass: organizationalUnit
ou: testadd
description: Ciao mondo! (how Italians say Hello world!)

adding new entry ou=testadd, o=sunblueprints
ldap_add: Operations error
ldap_add: additional info: Trigger SunBlueprintsBigTrigger forbids
the creation of entry ou=testadd,o=sunblueprints
```

It works; we receive the expected message because of the `IGNORE` action triggered by `SunBlueprintsBigTrigger`. Reviewing the log file shows the following:

CODE EXAMPLE 19 Verifying the Log File

```
[21/Oct/2003:02:05:57 +0200] - INFORMATION - (PID=4017,ThID=25625)
triggers_pre_add_fn  - conn=2 op=1 msgId=30 -  Entering
[21/Oct/2003:02:05:57 +0200] - INFORMATION - (PID=4017,ThID=25625)
triggers_pre_add_fn  - conn=2 op=1 msgId=30 -  Looking for
triggers...
[21/Oct/2003:02:05:57 +0200] - INFORMATION - (PID=4017,ThID=25625)
find_trigger  - conn=2 op=1 msgId=30 -  Entering(0x8352190,
0x47312008,
(&(&(on=ou=testadd,o=sunblueprints)(before=1))(enabled=true)))
[21/Oct/2003:02:05:57 +0200] - INFORMATION - (PID=4017,ThID=25625)
find_trigger  - conn=2 op=1 msgId=30 -  Getting results
[21/Oct/2003:02:05:57 +0200] - INFORMATION - (PID=4017,ThID=25625)
find_trigger  - conn=2 op=1 msgId=30 -  Returning
[21/Oct/2003:02:05:57 +0200] - INFORMATION - (PID=4017,ThID=25625)
triggers_pre_add_fn  - conn=2 op=1 msgId=30 -  Found triggers
ou=testadd,o=sunblueprints
...
```

The routine `triggers_pre_add_fn` is the add pre-operation function of our pre-operation plug-in. It calls `find_trigger` to search for triggers. When it finds one, it executes. As a counter proof, now let's delete the trigger, and re-add the test entry:

CODE EXAMPLE 20 Deleting the Trigger and Reading the Test Entry

```
TRIGGERS>> delete trigger SunBlueprintsBigTrigger;
.
 ------------- Server output ----------------
Statement #0
Error code: 0
Error msg: Trigger SunBlueprintsBigTrigger successfully deleted
 -----------------------------------------
TRIGGERS>> list all triggers;
.
 ------------- Server output ----------------
Statement #0
Error code: 0
Error msg: 1 triggers found
trgme
 -----------------------------------------
```

Try to add the entry again:

CODE EXAMPLE 21 Adding a Duplicate Entry

```
<sunone slapd-sunblueprints>~$ ldapmodify -p 10389 -D
"cn=Directory Manager" -w manager0
dn: ou=testadd, o=sunblueprints
changetype: add
objectclass: organizationalUnit
ou: testadd
description: Ciao mondo! (how Italians say for Hello world!)

adding new entry ou=testadd, o=sunblueprints
```

No problem now, because we deleted the trigger on this entry. Our language provides a construct to temporarily disable a trigger instead of removing it.

To disable a trigger, use the following as an example:

CODE EXAMPLE 22 Disabling a Trigger

```
TRIGGERS>> disable trigger SunBlueprintsBigTrigger;
.
 ------------- Server output ---------------
Statement #0
Error code: 0
Error msg: Trigger SunBlueprintsBigTrigger successfully disabled
 -----------------------------------------
```

If you repeat the LDAP add of the `ou=testadd` test entry, it will work. It works because the `triggers_pre_add_fn` function that is called before the internal add calls the `find_trigger` with a search filter such as the following:

```
(&(&(on=ou=testadd,o=sunblueprints)(before=0))(enabled=true)))
```

This filter searches expressly for only enabled triggers.

The LDAP triggers language has many other possibilities. Our code example implements only some of them. We invite you to discover the other features and functions.

Using LDAP Triggers in Production Environments

We've provided a good sampling of LDAP triggers and code samples for implementing them, but you might be asking "Are LDAP triggers ready to move from a developer's bench to a production environment?"

We suggest that you contemplate some preliminary considerations before moving to a production environment. Directory architects, and in a wider sense all computer system architects, think somewhat differently from developers. Architects are often responsible for designing complex architectures, where many components integrate and concur to provide global services. Therefore, their main focus is the whole application service offering, not just individual components. Before delivering your new fully featured component, you might want to make sure that it does not negatively impact any of the running services in terms of availability, scalability, service level agreement, recoverability, and so on.

To meet architects needs, we must in turn ask if our code will negatively impact Sun Java System Directory Server, the container in which the framework runs, and the directory architecture layout that it is plugged into. Consider the following:

- Do your plug-ins impact write-performance? Do they add extra time to normal write operations?
- Can you measure the impact?
- Is your solution scalable?
- Are your LDAP triggers suitable for a replication environment? If not, what countermeasures must be taken to make it work?
- Are your triggers secure?

During the delivery phase, these and other concerns could arise.

Our code example, as it is, is not scalable. It could impact the normal service level at more than a tolerable level. A refactoring is necessary. For example, we use a fixed-length array for storing the result of user statement parsing. We suppose that the user will not submit more than a certain number of commands; this can be recovered using lists instead of arrays. Another weakness is in the mechanism used to effectively trigger the action; a search is always implied, and this is not good.

Simple deployments, made of a single instance are fairly uncommon. Most often, you have at least a replication architecture, with one or more suppliers and many consumers (and hubs). There could even be more complex setups like eLDAP, where some instances are tricked to work as multiplexors (MUXs) so that operations are dispatched following a specified logic. In such environments, it is especially important that the trigger framework has some level of awareness of the underlying complexities. For example, in a MMR environment, the trigger must be aware of conflict resolution issues on update operations. Also, it must be made clear by the trigger framework designer what components of the architecture setups (supplier, consumer, hub, mux) the triggers should activate.

Despite these weakness, which can be corrected with not much work, LDAP triggers are overall a good thing. Their primary strength is their standard grammar. To add new features, you simply extend the language, add an action to the parser specification file, and implement the feature on the server side. With this process you can add more complex rules, beyond the basic ones introduced in the code example. For example, through extended trigger language, you can specify new instructions as follows:

- “`Change the attribute AAA of entry X when the attribute BBB of entry Y is changed`”
- “`Do this when the server starts /stop`”
- “`Replicate this change to my ORACLE when the this entry changes`”

About the Author

Nicola Venditti works at Sun Microsystems Italy as project engineer and is specialized on Sun ONE Directory Server. Before joining Sun in March, 2002 he spent four years at Informix where he worked as a middleware expert and database specialist.

Related Resources

Publications

Note – The following publications reference the Sun ONE products, which were recently renamed Sun Java System products. At this time, the previous titles are accurate. Newer publications will reflect the new naming.

- *Sun ONE Directory Server 5.2 Administration Guide.* `http://docs.sun.com/db/doc/816-6698-10`
- *Sun ONE Directory Server 5.2 Plug-In API Programming Guide* `http://docs.sun.com/db/doc/816-6702-10`
- *Sun ONE Directory Server 5.2 Plug-In API Reference* `http://docs.sun.com/db/doc/816-6701-10`
- *Sun ONE Server Console 5.2 Server Management Guide*, Sun Microsystems, Inc., June 2003, 816-6704-10.
- Venditti, Nicola. "Writing an Authentication Plug-in for a Sun ONE Directory Server," Sun BluePrints OnLine, March 2003. `http://www.sun.com/solutions/blueprints`

Web Sites

- Enhanced LDAP (eLDAP): `http://webhome.central/eLDAP`.
- Sun Java System (formerly Sun ONE) product documentation: `http://www.sun.com/documentation`.
- Sun BluePrints OnLine: `http://www.sun.com/blueprints`.

Ordering Sun Documents

The SunDocs℠ program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals through this program.

Accessing Sun Documentation Online

The `docs.sun.com` web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com/`

To reference Sun BluePrints OnLine articles, visit the Sun BluePrints OnLine Web site at: `http://www.sun.com/blueprints/online.html`