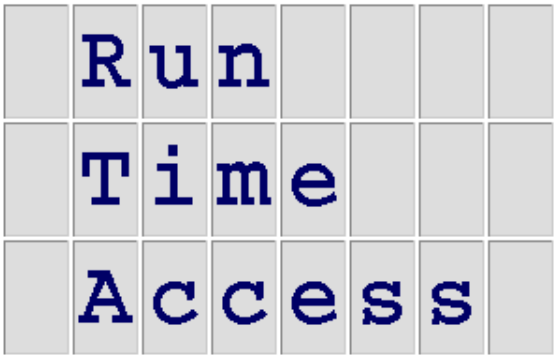
LinuxFocus article number 338
http://linuxfocus.org

Talking to a Running Process

by Bob Smith
<bob/at/linuxtoys.org>

About the author:

Bob is a Linux programmer and an electronics hobbyist. You can find his latest project at www.runtimeaccess.com and his homepage at www.linuxtoys.org.



Abstract:

Run Time Access is a library that lets you view the data structures in your program as tables in a PosgreSQL database or as files in a virtual file system (similar to /proc). Using RTA makes it easy to give your daemon or service several types of management interfaces such as web, shell, SNMP, or framebuffer.

10 Second Overview

Say you have a program with data in an array of structures. The structure and array are defined as:

```
struct mydata {
    char   note[20];
    int    count;
}

struct mydata mytable[] = {
  { "Sticky note", 100 },
  { "Music note", 200 },
  { "No note", 300 },
};
```

If you build your program with the Run Time Access library you will be able to examine and set the program's internal data from the command line or from another program. Your data appears as if it were in a PostgreSQL database. The following illustrates how you might use Bash and psql, the PostgreSQL command line tool, to set and read data in your program.

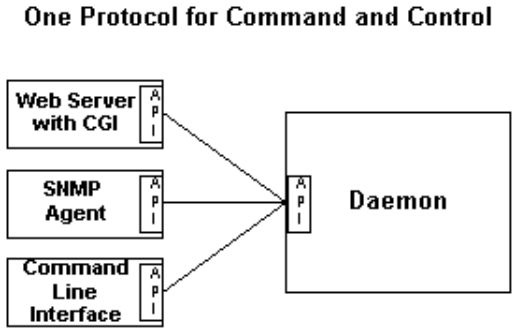
```
# myprogram  &

# psql -c "UPDATE mytable SET note = 'A note' LIMIT 1"
UPDATE 1

# psql -c "SELECT * FROM mytable"
   note     | count
------------+-------
 A note     | 100
 Music note | 200
 No note    | 300

#
```

This article explains why something like RTA is needed, how to use the RTA library, and what advantages you can expect from using RTA.

Many UIs -- One Service

Traditional UNIX communicated with a service by putting its configuration data into /etc/application.conf and its accumulated output in /var/log/application.log. This accepted approach is probably wrong for today's services that run on appliances and are configured by relatively untrained sys-admins. The traditional approach fails because we now want several types of simultaneous user interfaces, and we want each of those interfaces to exchange configuration, status, and statistics with the service while it is running. What is needed is run time access.

Newer services need many types of user interfaces and we developers may not be able to predict what interface will be needed most. What we need to do is to separate the user interface from the service using a common protocol and to build the user interfaces using the common protocol. This makes it easier to add interfaces when needed and the separation may make testing easier since each piece can be tested independently. We want an architecture something like this:



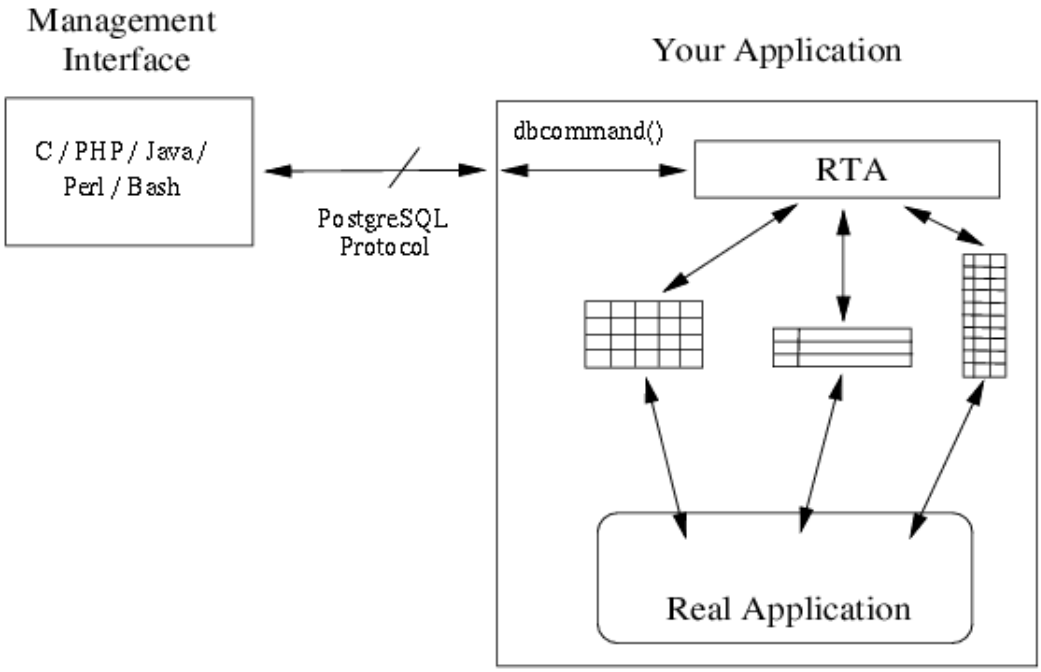
The types of user interface to consider include web, command line, framebuffer, SNMP, keypad and LCD, LDAP, native Windows, and other custom interfaces. Clearly a common API and protocol to all the user interfaces would be a good idea. But what kind of API and protocol?

A Database Interface

RTA chooses to use a PostgreSQL database as the common API and protocol. Configuration, state, and statistics are put into arrays of structures that appear on the API as tables in a PostgreSQL database. The user interface programs are written as clients connecting to a PostgreSQL database. This approach has two big benefits:

- The user interface clients use a well known, well documented, and well debugged API. Using PostgreSQL dramatically reduces development time. Also, PostgreSQL has binding for C, Java, PHP, Perl, and almost all other popular languages so you can program the UI in the language right for the job.
- The paradigm of *table in a database* matches fairly well how most of us write programs that provide a service. We use data structures for what would be *rows* and arrays or linked-list for what would be *tables*.

The RTA library is the glue that ties our arrays or linked lists of data structures to the PostgreSQL clients. The architecture of an application using RTA should look something like ...



Here we call it a *management interface* since it is intended for status, statistics, and configuration. Although only one interface is shown, you should remember that you can have many interfaces for your application, and they can all access the application simultaneously.

PosgreSQL uses TCP as the transport protocol so your application needs to be able to bind to a TCP port and accept connections from the various user interfaces. All bytes received from an accepted connection are passed into RTA using the dbcommand() subroutine. Any data to be returned to the client is in a buffer returned from dbcommand().

How does RTA know what tables are available? You have to tell it.

Defining Tables

You tell RTA about your tables with data structures and by calling the rta_add_table() subroutine. The TBLDEF data structure describes a table and the COLDEF structure describes a column. Here is an example that illustrates how to add a table to the RTA interface.

Say you have a data structure with a string of length 20 and an integer, and that you want to export a table with 5 of these structures. You can define the structure and table as follows:

```
struct myrow {
    char    note[20];
    int     count;
};

struct myrow mytable[5];
```

Each field in the `myrow` data structure is a column in a database table. We need to tell RTA the name of the column, what table it is in, its data type, its offset from the start of the row, and whether or not it is read-only. We can also define *callback* routines which are called before the column is read and/or after it is written. For our example we will assume that `count` is read-only and that we want `do_note()` called whenever there is a write to the `note` field. We build an array of COLDEF which is added to the TBLDEF and which has one COLDEF for each structure member.

```
COLDEF mycols[] = {
    {
        "atable",          // table name for SQL
        "note",            // column name for SQL
        RTA_STR,           // data type of column/field
        20,                // width of column in bytes
        0,                 // offset from start of row
        0,                 // bitwise OR of boolean flags
        (void (*)()) 0,    // called before read
        do_note(),         // called after write
        "The last field of a column definition is a string "
        "to describe the column.  You might want to explain "
        "what the data in the column means and how it is "
        "used."},
    {
        "atable",          // table name for SQL
        "count",           // column name for SQL
        RTA_INT,           // data type of column/field
```

```
            sizeof(int),       // width of column in bytes
            offsetof(myrow, count),  // offset from start of row
            RTA_READONLY,      // bitwise OR of boolean flags
            (void (*)()) 0,    // called before read
            (void (*)()) 0,    // called after write
            "If your tables are the interface between the user "
            "interfaces and the service, then the comments in "
            "column and table definitions form the functional "
            "specification for your project and may be the best "
            "documentation available to the developers."
    };
```

Write callbacks can be the real engine driving your application. You may want to have changes to a table trigger other changes or a reconfiguration of your application.

You tell RTA about tables by giving it the name of the table, the length of each row, an array of COLDEFS to describe the columns, a count of the columns, the name of the save file if you want some of the columns to be non-volatile, and a string to describe the table. If the table is a static array of structs you give the starting address and the number of rows in the table. If the table is implemented as a linked list you give RTA a routine to call to *iterate* from one row to the next.

```
TBLDEF   mytableDef = {
      "atable",                 // table name
      mytable,                  // address of table
      sizeof(myrow),            // length of each row
      5,                        // number of rows
      (void *) NULL,            // iterator function
      (void *) NULL,            // iterator callback data
      mycols,                   // Column definitions
      sizeof(mycols / sizeof(COLDEF), // # columns
      "",                       // save file name
      "A complete description of the table."
};
```

Normally you would want the table name as seen by SQL to be the same as its name inside the program. The example switched from `mytable` to `atable` just to show that the names do not need to be the same.

Given all of the above code, you can now tell RTA about your table.

```
    rta_add_table(&mytableDef);
```

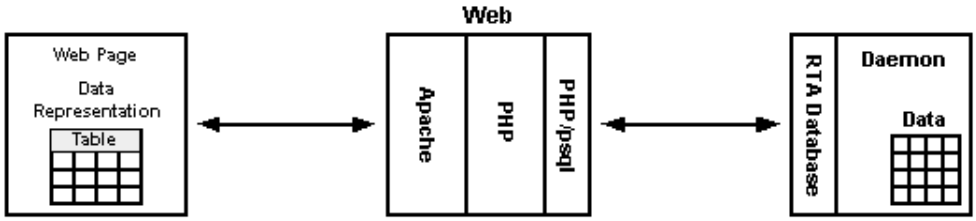
That's all there is to it. To use RTA you need to learn how to use two data structures (COLDEF and TBLDEF) and two subroutines (dbcommand() and rta_add_table()).

The above code is meant to give you a taste of how RTA works. It is not meant as a full tutorial or complete working example. A complete working example and a full full description of the RTA API and data structures is on the RTA web site (www.runtimeaccess.com).

Just as you define tables for use in your application, so RTA defines its own set of internal tables. The two most interesting of these table are rta_tables and rta_columns which are, of course, tables to list and describe all of the tables and columns you've defined. These are the so-called *system tables*. The system tables do for a database what `ls` does for a file system and `getnext()` does for SNMP.

The Table Editor

One of the utilities that ships with RTA is a small PHP program that uses the system tables to list your RTA tables in a web browser window. The table names are links and clicking on a table name displays the first 20 rows of the table. If the table has any editable fields you can click on a row to open up an edit window for that row. All this is done using the table and column descriptions found in the system tables. The data flow is depicted in the following diagram.



The top level view of the table editor display for the RTA sample application is shown below.

RTA Table Editor

Table Name	Description
rta_tables	The table of all tables in the system. This is a pseudo table and not an array of structures like other tables.
rta_columns	The list of all columns in all tables along with their attributes.
pg_user	The table of Postgres users. We spoof this table so that any user name in a WHERE clause appears in the table as a legitimate user with no super, createDB, trace or catupd capability.
rta_dbg	Configure of debug logging. A callback on the 'target' field closes and reopens syslog(). None of the values in this table are saved to disk. If you want non-default values you need to change the rta source or do an SQL_string() to set the values when you initialize your program.
rta_stat	Usage and error counts for the rta package.
mytable	A sample application table
UIConns	Data about TCP connections from UI frontend programs

By the way, if all has gone well in the publishing of this LinuxFocus article, the table names given above should have live links to the sample application running on the RTA web server in Santa Clara,

California. A good link to follow is the `mytable` link.

Two Commands

Run Time Access is a library that links management or user interface programs written with the PostgreSQL client library (libpq) to your application or daemon. RTA is an interface, not a database. As such, it needs only two SQL commands, SELECT and UPDATE.

The syntax for the SELECT statement is:

```
SELECT column_list FROM table [where_clause] [limit_clause]
```

The column_list is a comma separated list of column names. The where_clause is an AND separated list of comparisons. The comparison operators are =, |=, >=, <=, >, and <. A limit_clause has the form `[LIMIT i] [OFFSET j]`, where `i` is the maximum number of rows to return and we skip over `j` rows before starting output. Some examples might help clarify the syntax.

```
SELECT * FROM rta_tables

SELECT notes, count FROM atable WHERE count > 0

SELECT count FROM atable WHERE count > 0 AND notes = "Hi Mom!"

SELECT count FROM atable LIMIT 1 OFFSET 3
```

Setting the LIMIT to 1 and specifying an OFFSET is a way to get a specific row. The last example above is equivalent to the C code `(mytable[3].count)`.

The syntax of the UPDATE statement is:

```
UPDATE table SET update_list [where_clause] [limit_clause]
```

The where_clause and limit clause are as described above. The update_list is a comma separated list of column assignments. Again, some examples will help.

```
UPDATE atable SET notes = "Not in use" WHERE count = 0

UPDATE rta_dbg SET trace = 1

UPDATE ethers SET mask = "255.255.255.0",
                  addr = "192.168.1.10"
              WHERE name = "eth0"
```

RTA recognizes both upper and lower case reserved words although the examples here use upper case for all of the SQL reserved words.

Download and Build

You can download RTA from its web site at www.runtimeaccess.com (copyright of RTA is LGPL). Be careful in selecting which version of RTA to download. The latest RTA version uses the newer PostgreSQL protocol introduced with the 7.4 version of PostgreSQL. Most current Linux distributions use the 7.3 version. While you can use an older version of RTA for initial trials you should use the latest version to get the latest bug fixes and enhancements.

Untarring the package should give you the following directories:

```
./doc            # a copy of the RTA web site
./empd           # a prototype deamon built with RTA
./src            # source files for the RTA library
./table_editor   # PHP source for the table editor
./test           # source for a sample application
./util           # utilities used in writing RTA
```

Thanks to Graham Phillips, the 1.0 version of RTA has autoconf support. Graham ported RTA from Linux to Mac OS X, Windows, and FreeBSD. Using the 1.0 release you can build RTA with the usual

```
./configure
make
make install      # (as root)
```

The installation puts librtadb.so and the associated library files in the /usr/local/lib directory. To use RTA you can add this directory to /etc/ld.so.conf and running the ldconfig command, or you can add the directory to your loader path with:

```
export LD_LIBRARY_PATH=/usr/local/lib
```

The installation puts the RTA header file, rta.h, in /usr/local/include.

The make builds a test program in the `test` directory and you can test your installation by changing directory to the test directory and running `./app &`. A `netstat -nat` should show a program listening on port 8888. Now you can run psql and issue SQL commands against your test application.

```
cd test
./app &

psql -h localhost -p 8888
Welcome to psql 7.4.1, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

# select name from rta_tables;
    name
-------------
```

```
 rta_tables
 rta_columns
 rta_dbg
 rta_stat
 mytable
 UIConns
(6 rows)
```

While it looks like you are connected to a database, you are not. Don’t forget: the only two commands you can use are SELECT and UPDATE.

Advantages of RTA

The advantages of separating the user interface programs from the daemon proper fall into the broad categories of design, coding, debug, and capabilities.

From a design point of view, the division forces you to decide early in the design what exactly is offered as part of the UI without worrying how it is displayed. The thought process required to design the tables forces you to think through the real design of your application. The tables might form the internal functional specification of your application.

While coding, the table definitions are what the daemon engineers build to and what the UI engineers build from. The division of UI and daemon means you can hire UI experts and daemon experts independently and they can code independently which might help bring the product to market sooner. Since there are Postgres bindings for PHP, Tcl/Tk, Perl, and "C", your developers can use the right tool for the job.

Debug is faster and easier because both the UI and the daemon engineers can simulate the other half easily. For example, the UI engineers could run their UI programs against a real Postgres DB which has the same tables the daemon will have. Testing the daemon can be easier and more complete since it is easy to build test scripts to simulate the UI, and it is easy to examine internal status and statistics while a test runs. The ability to force an internal state or condition helps test corner-cases which are sometimes difficult to do in a lab setup.

The capability of your product can be expanded with RTA. Your customers will really appreciate being able to see detailed status information and statistics while the program is running. Separating the UIs from the daemon means you can have more UI programs: SNMP, command line, web, LDAP, and the list goes on. This flexibility will be important to you if (when!) your customers ask for custom UIs.

RTA offers several other features you might want in a package of this type:

- Application data model reflected by the API data model
- Remote access to the application
- Use of standards and existing software by the application
- Few new protocols and APIs to learn

- Discovery mechanisms for the application
- Few constraints on the application
- Resource locking
- CPU and memory efficiency

Summary

This article has presented a very brief introduction to the RTA library and its capabilities. The RTA web site has a FAQ, a complete description of the API, and several sample client programs.

Just as RTA can make your data structures visible as tables in a database, so it can make them visible as files in a virtual file system. (Using the File System in Userspace (FUSE) package by Miklos Szeredi.) The web site has more information on how to use the file system interface.
