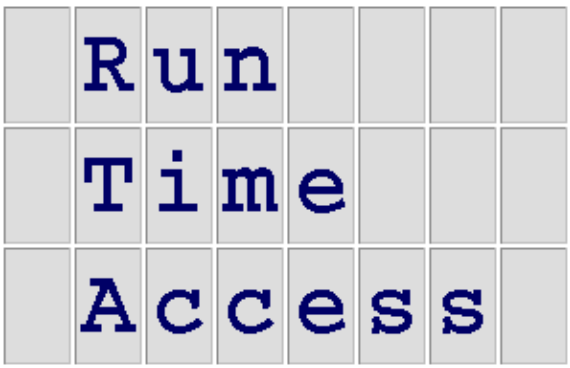
Kommunizieren mit einem laufenden Prozess

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

About the author:

Bob ist Linuxprogrammierer, sein Hobby ist Elektronik. Wir finden sein neuestes Projekt unter http://www.runtimeaccess.com/ und seine Homepage ist www.linuxtoys.org.



Abstract:

Run Time Access ist eine Bibliothek, die erlaubt mit den Datenstrukturen unserer Programme– in Form von Tabellen in einer PosgreSQL–Datenbank– oder als ein virtuelles Dateisystem (ähnlich wie /proc) zu kommunizieren. Die Anwendung von RTA erleichtert unserem Dämon oder Service mehrere Arten von Managementoberflächen, wie Web, Shell, SNMP oder Framebuffer.

Kurzüberblick in 10 Sekunden

Nehmen wir an, wir haben ein Programm, dessen Daten in einer Arraystruktur angeordnet sind. Die Struktur und das Array sind wie folgt definiert:

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

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

Bauen wir unser Programm mit der Run Time Access–Bibliothek, ist es möglich, die internen Daten des Programms mittels Kommandozeile, bzw. über ein anderes Programm, einzusehen oder zu beeinflussen. Unsere Daten erscheinen, als wären sie Teil einer PosgreSQL–Datenbank. Nachfolgend zeigen wir, wie wir mit Bash und psql – dem PosgreSQL–Kommandozeilenwerkzeug – Daten in unserem Programm bearbeiten und eingeben können.

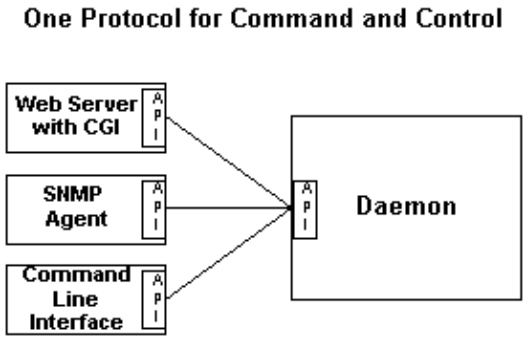
```
# myprogram  &

# 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

#
```

Dieser Artikel erklärt, warum so etwas wie RTA benötigt wird, wie man die RTA−Bibliothek benutzt und welche Vorteile wir vom Gebrauch von RTA erwarten können.

Viele Benutzeroberflächen – ein Service

Herkömmlich kommuniziert UNIX mit einem Service, indem es seine Konfigurationsdaten in */etc/application.conf* und den zusammengefassten Output in */var/log/application.log* unterbringt. Diese gebräuchliche Vorgehensweise ist wahrscheinlich falsch für die heutigen Services, die mittels Dienstgeräten von relativ untrainierten Systemadministratoren durchgeführt werden. Das traditionelle Vorgehen versagt jedoch, da wir heute mehrere Benutzeroberflächen gleichzeitig bereitstellen wollen − ausserdem wollen wir mit diesen Oberflächen Konfigurationen, Status und Statistiken im laufenden Betrieb mit dem Service austauschen. Was wir benötigen ist Laufzeitzugriff.

Die neueren Services benötigen viele Arten von Benutzeroberflächen, und wir Entwickler können oft nicht vorhersehen, welche Oberflächen am häufigsten benutzt werden. Daher müssen wir die Benutzeroberfläche vom Service mit einem gebräuchlichen Protokoll trennen, mit diesem müssen wir die Benutzeroberflächen bauen. Das erleichtert das Hinzufügen von Benutzeroberflächen nach Bedarf, diese Trennung erleichtet deren unabhängiges Testen. Wir benötigen eine Architektur die etwa so aussieht:



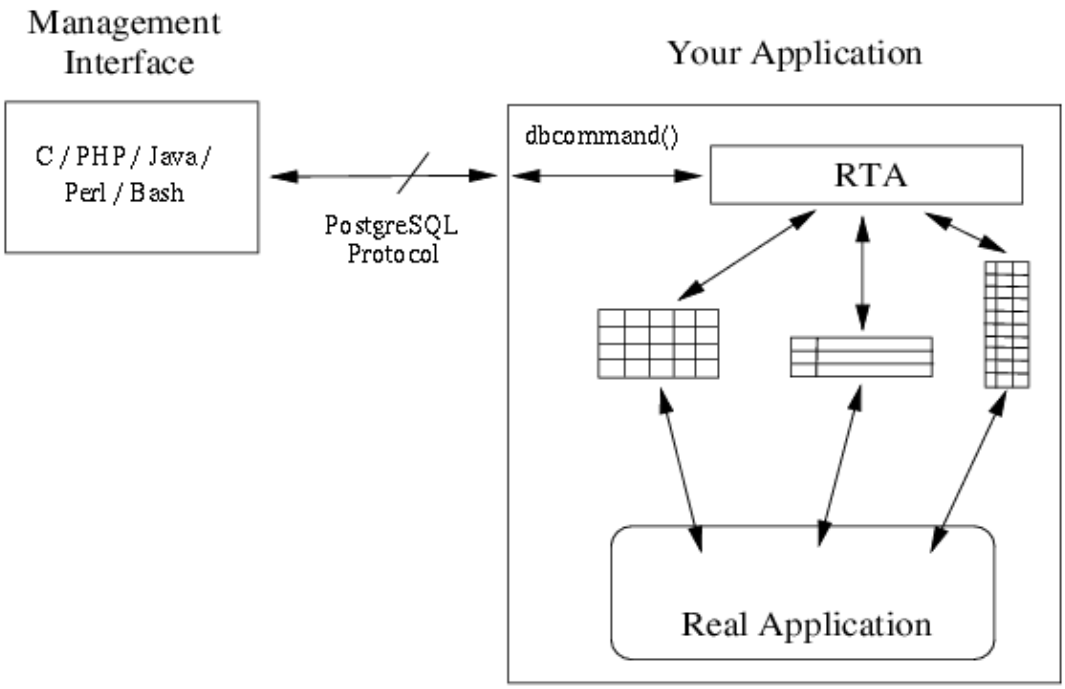
Die möglichen Oberflächentypen umfassen Web, Befehlszeile, Einzelbildpuffer, SNMP, Tastatur und LCD, LDAP, native Windows und andere benutzerspezifische Oberflächen. Aber welches API und Protokoll?

Eine Datenbank−Benutzeroberfläche

RTA benutzt eine PosgreSQL−Datenbank als Standard−API und −Protokoll. Konfiguration, Status und Messwerte werden in Strukturarrays gelegt, die im API als Tabellen in einer PosgreSQL−Datenbank erscheinen. Die Benutzeroberflächen sind als Clients aufgebaut, die sich mit einer PosgreSQL−Datenbank verbinden. Dieses Vorgehen hat zwei grosse Vorteile:

- Die Benutzeroberflächen−Clients verwenden eine Anwendungsprogrammierschnittstelle (API), die weitverbreitet, gut dokumentiert und debugt ist. Die Anwendung von PosgreSQL verkürzt die Entwicklungszeit dramatisch. PosgreSQL hat Bindung für C, Java, PHP, Perl und fast alle anderen gebräuchlichen Sprachen, das ermöglicht das Programmieren der Benutzeroberfläche in der für die Aufgabe am besten geeigneten Sprache.
- Das Paradigma *Tabelle in einer Datenbank* bezeichnet ziemlich gut, wie die meisten von uns Serviceprogramme schreiben. Wir benutzen Datenstrukturen in den *Zeilen* und Arrays oder Linklisten in den *Spalten* in was wir als *Tabellen* bezeichnen würden.

Die RTA−Bibliothek verbindet unsere Arrays oder Linklisten von Datenstrukturen mit den PosgreSQL−Clienten. Die Architektur einer Anwendung, die RTA benutzt, sollte ungefär so aussehen...



Wir nennen das hier *Managementoberfläche*, da sie für Status, Statistiken und zur Konfiguration gedacht ist. Obwohl nur eine Oberfläche gezeigt wird, sollten wir nicht vergessen, dass unsere Anwendung viele Oberflächen haben kann, welche die Anwendung gleichzeitig ansprechen können.

PosgreSQL benutzt TCP als Transportprotokoll, d.h. unsere Anwendungen müssen in der Lage sein, an einen TCP−Port zu binden und Verbindungen von den verschiedenen Benutzeroberflächen zu erlauben. Alle Bytes von erlaubten Verbindungen werden mit der *dbcommand()*−Unterroutine ins RTA weitergeleitet. Alle Daten, die durch *dbcommand()* an den Client zurückgeleitet werden, sind in einem Puffer.

Wie erkennt RTA welche Tabellen vorhanden sind? Wir müssen es ihm mitteilen.

Tabellen definieren

Wir teilen RTA mit, wo unsere Tabellen mit den Datenstrukturen zu finden sind und führen die *rta_add_table()*–Unterroutine durch. Die TBLDEF–Datenstruktur beschreibt eine Tabelle und COLDEF eine Spalte. Hier ein Beispiel, wie man eine Tabelle der RTA–Oberfläche hinzufügt.

Nehmen wir an, wir haben einen String der Länge 20 und einen Integer. Wir wollen eine Tabelle mit fünf dieser Strukturen exportieren. Die Struktur und die Tabelle definieren wir wie folgt:

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

struct myrow mytable[5];
```

Jedes Feld in der `myrow`–Datenstruktur ist eine Spalte in einer Datenbanktabelle. Wir müssen RTA den Namen der Spalte, in welcher Tabelle, den Datentyp, den Offset vom Anfang der Zeile und ob diese schreibgeschützt ist, mitteilen. Wir können auch *callback*–Routinen definieren, die aufgerufen werden, bevor eine Spalte gelesen oder nachdem in diese geschrieben wurde. In unserem Beispiel nehmen wir an, dass `count` schreibgeschützt ist und dass `do_note()` nach jedem Schreiben im `note`–Feld aufgerufen wird. Wir bauen ein Array von COLDEF, welches zu TBLDEF hinzugefügt wird und das nur ein COLDEF für jedes Strukturglied hat.

```
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."
```

```
    };
```

Schreibrückrufe können der wahre Antrieb unserer Anwendung sein. Wir könnten auch fordern, dass Änderungen an einer Tabelle andere Änderungen oder Rekonfiguration unserer Anwendung hervorrufen.

Wir informieren RTA über unsere Tabellen, indem wir folgendes definieren: deren Namen, die Länge jeder Zeile, das Array von COLDEF – um die Spalten zu beschreiben, die Anzahl der Spalten, den Namen der Speicherdatei – falls wir einige Spalten nichtflüchtig wollen – und einen String, um die Tabelle zu beschreiben. Besteht die Tabelle aus einem statischen Array von Strukturen, bezeichnen wir die Startadresse und die Zeilenanzahl der Tabelle. Wird die Tabelle als verlinkte Liste ausgeführt, lassen wir RTA die Routine *iterate* von Reihe zu Reihe ausführen

```
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."
};
```

Normalerweise würde der Tabellennamen, den wir SQL angeben, der gleiche sein, der im Programm erscheint. Im Beispiel änderten wir diesen von `mytable` in `atable`, um zu demonstieren, dasss der Namen nicht gleich sein muss.

Mit dem oben aufgeführten Code informieren wir nun RTA über unsere Tabelle

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

Das wär's denn. Um RTA zu benutzen, müssen wir die Anwendung von zwei Datenstrukturen (COLDEF und TBLDEF), sowie von zwei Unterroutinen (*dbcommand()* und *rta_add_table())* lernen.

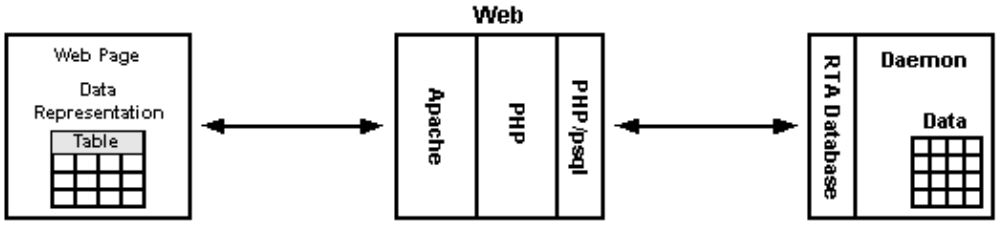
Der vorangegangene Code soll einen Geschmack davon vermitteln, wie RTA arbeitet. Er ist nicht als umfassendes Tutorial oder vollständiges Arbeitsbeispiel gedacht – dieses finden wir, nebst einer vollständigen Beschreibung der RTA API und der Datenstrukturen auf der RTA–Webseite (http://www.runtimeaccess.com/).

Ebenso wie wir Tabellen für unsere Anwendung definieren, erzeugt RTA seine eigene Reihe von internen Tabellen. Die zwei interessantesten dieser Tabellen sind die *rta_tables* und *rta_columns*, das sind natürlich die zu verlinkenden Tabellen, sie beschreiben alle Tabellen und Spalten, die wir definiert haben. Es sind die sogenannten *system tables*. Die Systemtabellen sind für eine Datenbank was `ls` für das Dateisystem und `getnext()` für SNMP sind.

Der Tabelleneditor

Eine der Utilities, die mit RTA geliefert wird, ist ein kleines PHP–Programm, welches die Systemtabellen

benutzt, um unsere RTA−Tabellen in einem Webbrowserfenster zu verlinken. Die Tabellennamen sind Links , durch Klicken auf den Tabellennamen werden die ersten 20 Zeilen der Tabelle dargestellt. Hat die Tabelle editierbare Felder, können wir durch Klicken auf eine Zeile ein Editfenster für diese Zeile öffnen. All das wird durch die Anwendung der Tabellen− und Spaltenbeschreibungen aus den Systemtabellen durchgeführt. Der Datenfluss ist im folgenden Diagramm dargestellt.



Das Display der Toplevelansicht des Tabelleneditors für die Anwendung des RTA−Beispiels ist nachfolgend gezeigt.

RTA Tabelleneditor

Tabellenname	Beschreibung
rta_tables	Die Tabelle aller Tabellen im System. Das ist eine Pseudotabelle und kein Strukturarray wie andere Tabellen.
rta_columns	Die Liste aller Spalten aller Tabellen, einschliesslich deren Attributen.
pg_user	Die Tabelle der Posgresbenutzer. Wir täuschen diese Tabelle, sodass jeder Benutzernamen in einer '*where_clause*' in der Tabelle als legitimer Benutzer erscheint, ohne Super−, createDB−, trace− oder catupd−Möglichkeit.
rta_dbg	Konfiguration des Debuglogging. Ein Rückruf auf das 'target'−Feld schliesst und öffnet *syslog()*. Keiner dieser Werte in diese Tabelle wird der Festplatte gesichert. Wünschen wir andere Werte, als die Grundeinstellung, müssen wir die RTA−Quelle ändern oder mit einen SQL−String die Werte beim Starten des Programms einstellen.
rta_stat	Benutzung− und Fehlerzählung des RTA−Pakets
mytable	Tabelle eines Anwendungsbeispiels
UIConns	Daten über TCP−Verbindungen der Benutzeroberfläche des Front−Ends−Programms.

Wenn alles planmässig mit der Veröffentlichung dieses LinuxFocus−Artikels gelaufen ist, sollten die Tabellennamen oben Links zum Anwendungsbeispiel auf dem RTA−Webserver in Santa Clara, California haben. Ein gut zu folgender Link ist `mytable`.

Zwei Befehle

Run Time Access ist eine Bibliothek, welche das Management– oder Benutzeroberflächen–Programm – geschrieben mit der PosgreSQL–Bibliothek (libpq) – zu unserer Anwendung oder Dämon linkt. RTA ist eine Oberfläche, keine Datenbank. Als solche benötigt sie nur zwei SQL–Befehle: SELECT und UPDATE.

Die Syntax für die SELECT–Eingabe ist:

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

Column_list ist eine Trennzeichen getrennte Liste der Spaltennamen. *Where_clause* ist eine UND getrennte Vergleichsliste. Die Vergleichsoperatoren sind =, |=, >=, <=, >, und <. *Limit_clause* sieht so aus: `[LIMIT i] [OFFSET j]`, hierbei ist `i` die maximale Anzahl der auszugebenden Zeilen, wir überspringen `j` Zeilen, ehe wir mit der Ausgabe beginnen. Mit einigen Beispielen wollen wir die Syntax erklären.

```
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
```

Mit der Wahl des LIMIT auf 1 und der Angabe des OFFSET erhalten wir eine bestimmte Zeile. Das letzte Beispiel oben ist äquivalent mit dem C–Code `(mytable[3].count)`.

Die Syntax des UPDATE–Statement ist:

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

Die *where_clause* und die *limit_clause* sind oben beschrieben. Die *update_list* ist eine kommagetrennte Liste von Spaltenbestimmungen. Einige Beispiele werden auch hier wieder helfen

```
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 erkennt die Gross–und Kleinschreibung von Schlüsselwörtern, unsere Beispiele hier benutzen alle Grossschreibung für die SQL–Schlüsselwörter.

Download und Build

Wir können RTA von seiner Webseite herunterladen http://www.runtimeaccess.com/ (das Copyright für RTA ist GPL). Bei der Auswahl des RTA–Downloads bitte auf die Version achten. Die jüngste RTA–Version benutzt das neuere PosgreSQL–Version 7.4 –Protokoll. Die meisten Linuxdistributionen benutzen die

Version 7.3. RTA kann mit einer ältere Version benutzt werden, zum Ausprobieren sollten wir jedoch die neueste Version mit den letzten Fehlerbehebungen und Verbesserungen benutzten.

Nach Anwendung von *untar* sollte das Packet folgende Verzeichnisse erzeugen:

```
./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
```

Dank den Bemühungen von Graham Phillips hat die RTA−Version 1.0 *autoconf*−Unterstützung. Graham portierte RTA von Linux auf Mac OS X, Windows und FreeBSD. Mit Release 1.0 können wir RTA wie üblich bauen

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

Mit der Installation werden *libtadb.so* und die zugehörigen Bibliothekdateien im */usr/local/lib*−Verzeichnis abgelegt. Um RTA zu benutzen, können wir dieses Verzeichnis zu */etc/ld.so.conf* hinzufügen und den Befehl *Idconfig* ausführen, oder wir können unser Verzeichnis in den Ladepfad einfügen:

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

Die Installation fügt die RTA−Headerdatei (rta.h) in */usr/local/include*.

Make baut ein Testprogramm im `test`−Verzeichnis. Wir können unsere Installation testen, indem wir in das Testverzeichnis wechseln und `./app &` ausführen. Ein `netstat -nat` sollte einen Programmlisting am Port 8888 zeigen. Damit können wir in unserer Testanwendung psql ausführen und SQL−Befehle eingeben.

```
cd test
./app &

psql -h localhost -p 8888
Welcome to psql 7.4.1, the PosgreSQL 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)
```

Es sieht aus, als wären wir direkt mit einer Datenbank verbunden, das trifft jedoch nicht zu. Wir sollten nicht vergessen: die einzigen Kommandos, die wir benutzen können sind SELECT und UPDATE.

Vorteile von RTA

Die Vorteile, die Benutzeroberflächen−Programme von der Dämonzugehörigkeit zu trennen, liegen im weiteren Bereich von Design, Coding, Debug und den Fähigkeiten.

Aus der Sicht des Design zwingt uns die Trennung – am Beginn des Designprozesses – zu der frühen Entscheidung, was genau in der Benutzeroberfläche geboten werden soll, ohne darauf einzugehen, wie das aussieht. Der Denkprozess für den Entwurf der Tabellen zwingt uns, den endgültigen Entwurf unserer Anwendung durchzudenken. Die Tabellen entstehen als die interne Spezifikation der Funktionen unserer Anwendung.

Während des Codierens dienen die Tabellen den Dämonentwicklern als Zielspezifikation und den Entwicklern der Benutzeroberflächen als Grundlage. Die Trennung in Benutzeroberfläche und Dämonen bedeutet, man kann Entwicklungsexperten unabhängig von einander anstellen und codieren lassen, damit kann das Produkt schneller auf den Markt gebracht werden. Da Posgresbindungen zu PHP, TCL/Tk, Perl und "C" bestehen, haben unsere Entwickler die angemessenen Werkzeuge für die Aufgabe.

Debug ist schneller und einfacher, da die Entwickler der Benutzeroberflächen und der Dämonen einander simulieren können. Die Entwickler der Benutzeroberfläche können zum Beispiel ihr Programm mit einer existierenden Posgresdatenbank prüfen, die auch vom Dämon benutzt wird. Das Testen des Dämon kann einfacher und vollständiger sein, da es einfach ist, Testscripts zur Simulation der Benutzeroberfläche zu bauen. Es ist auch einfach, den internen Status und Statistiken während des Tests zu untersuchen. Die Möglichkeit einen internen Status oder Bedingung herbeizuführen, hilft beim Testen von Grenzfällen, die im Labor oft schwierig durchzuführen sind.

Die Fähigkeiten unseres Produkts können mit RTA erweitert werden. Unsere Kunden werden für Detailstatusinformationen während des laufenden Programms echt dankbar sein. Die Abtrennung der Benutzeroberflächen von den Dämonen bedeutet, dass mehr Benutzeroberflächen zur Verfügung stehen können: SNMP, Befehlszeile, Web, LDAP – die Liste lässt sich verlängern. Diese Flexibilität ist wichtig, wenn Kunden massgeschneiderte Benutzeroberflächen wünschen.

RTA bietet mehrere andere wünschenswerte Eigenschaften:

- Dasdatenmodell der Anwendung entspricht dem API−Datenmodell
- Fernzugriff auf die Anwendung
- Gebrauch von Standards und existierender Software durch die Anwendung
- Wenige neue Protokolle und APIs zu lernen
- Suchmechanismen für die Anwendung
- Wenige Einschränkungen der Anwendung
- Ressourcensperre
- CPU – und Speicherleistung

Zusammenfassung

Dieser Artikel gab eine sehr kurze Einführung in die RTA−Bibliothek und deren Fähigkeiten. Auf der RTA−Webseite finden FAQ, eine vollständige Beschreibung der API und mehrere Beispiele von Clientprogrammen.

Genau wie RTA unsere Datenstrukturen als Tabellen einer Datenbank sichtbar macht, kann es diese als Dateien in einem virtuellen Dateisystem zeigen (mittels File System in Userspace (FUSE)−Paket von Miklos Szeredi.).

2005−01−11, generated by lfparser_pdf version 2.51