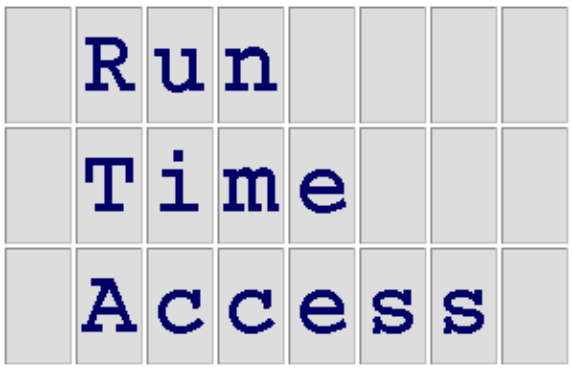
Dialoguer avec un processus en fonctionnement

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

L´auteur:

Bob est un programmeur Linux et un amateur d'électronique. Vous pouvez trouver son dernier projet sur www.runtimeaccess.com et sa page d'accueil sur www.linuxtoys.org.

Traduit en Français par:
Iznogood
<iznogood/at/iznogood–factory.org>



Résumé:

Run Time Access est une bibliothèque qui vous permet de voir les structures de données dans votre programme comme des tables dans une base de données PostgreSQL ou comme des fichiers dans un système de fichiers virtuel (similaire à /proc). L'utilisation de RTA facilite l'utilisation de types d'interfaces de gestion pour vos démons ou services tels que web, shell, SNMP ou framebuffer.

Survol en 10 secondes

Disons que vous avez un programme avec des données dans un tableau de structures. Cette structure et ce tableau sont définis comme :

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

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

Si vous construisez votre programme avec la bibliothèque Run Time Access, vous aurez la possibilité d'examiner et d'initialiser les données internes du programme depuis la ligne de commande ou depuis un autre programme. Vos données apparaissent comme si elles étaient dans une base de données PostgreSQL. Ce qui suit illustre la manière dont vous pouvez utiliser Bash et psql, l'outil en ligne de commande PostgreSQL, pour initialiser et lire les données dans votre programme.

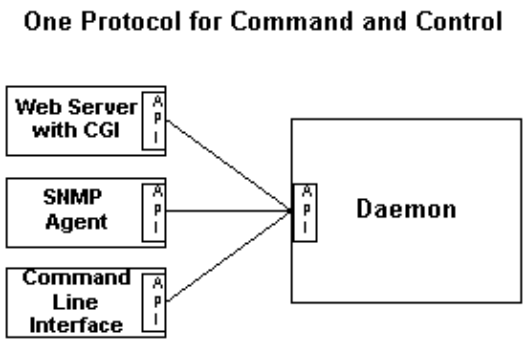
```
# myprogram  &

# 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

#
```

Ce article explique pourquoi RTA est nécessaire, comment utiliser la bibliothèque RTA et quels avantages vous pouvez attendre de l'utilisation de RTA.

Plusieurs interfaces utilisateurs –– Un Service

Les Unix traditionnels communiquent avec un service en mettant ses données de configuration dans /etc/application.conf et ses informations de sortie dans /var/log/application.log. Cette approche admise est probablement mauvaise pour les services actuels qui fonctionnent sur des appareils et qui sont configurés par des administrateurs système non entrainés. L'approche traditionnelle échoue car nous voulons maintenant plusieurs types d'interfaces utilisateurs simultanés et nous voulons qu'elles échangent leur configuration, leur statut et les services en cours de marche. Ce qui est nécessaire est un accès pendant le fonctionnement.

Les nouveaux services peuvent nécessiter plusieurs types d'interfaces utilisateurs et nous, les développeurs, ne pouvons prédire quelle interface sera la plus utilisée. Nous pouvons séparer l'interface utilisateur du service en utilisant un protocole et construire les interfaces utilisateurs en utilisant le protocole commun. Ceci facilite l'ajoût d'interfaces l'utilisant lorsque c'est nécessaire et la séparation facilite la phase de test car chaque morceau peut être testé indépendemment. Nous voulons une architecture qui ressemble à ceci:



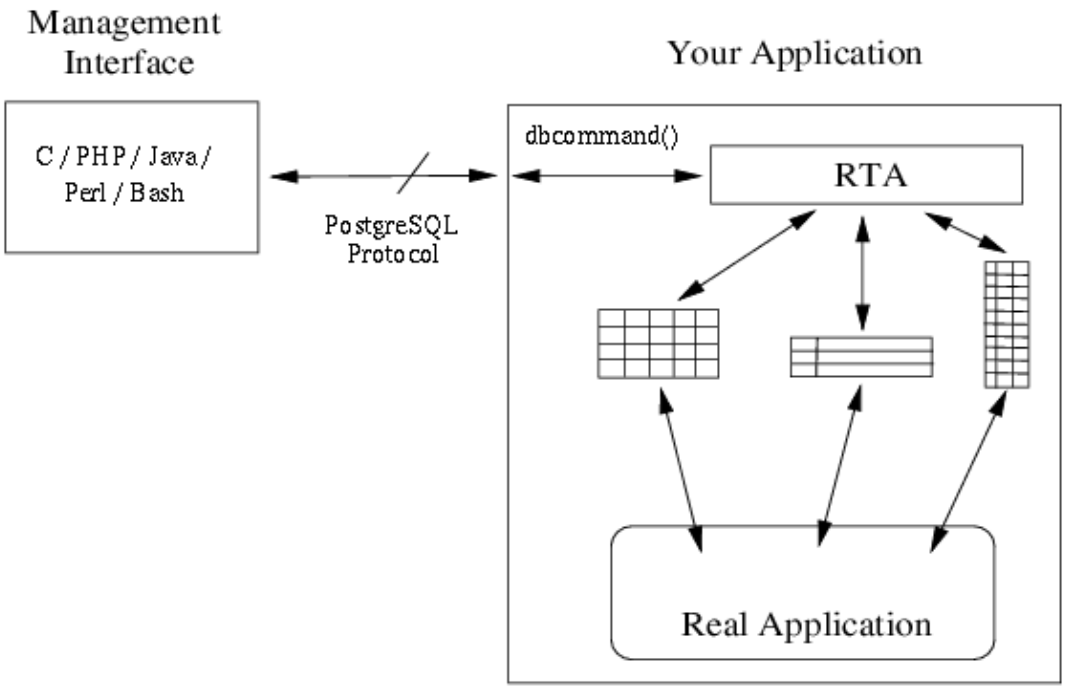
Les types d'interface utilisateur considérés incluent le web, la ligne de commande, framebuffer, SNMP, le clavier et l'écran sensitif, LDAP, Windows en natif et d'autres interfaces personnalisées. Une API et un protocole commun à toutes les interfaces utilisateur seraient, clairement, une bonne idée. Mais quel type d'API et de protocole?

Une interface de base de données

RTA choisit d'utiliser une base de données PostgreSQL comme API commune et protocole. La configuration, l'état et les statistiques sont incluse dans des tableaux de structures qui apparaissent sur l'API comme tables dans une base de données PostgreSQL. Les programmes de l'interface utilisateur sont écrits comme des clients connectés à une base de données PostgreSQL. Cette approche a deux grands avantages:

- Les clients de l'interface utilisateur utilisent une API bien connue, bien documentée et bien déboguée. L'utilisation de PostgreSQL réduit drastiquement le temps de développement. PostgreSQL a aussi des liaisons avec du C, Java, PHP, Perl et è peu près tous les autres langages populaires de telle manière que vous puissiez programmer l'interface utilisateur dans le langage adapté au travail.
- Le paradigme d'une *table dans une base de données* correspond assez bien à la manière dont la majorité d'entre nous écrit des programmes qui fournissent un service. Nous utilisons les structures de données pour ce qui pourrait être des *lignes* et les tableaux ou les listes liées pour ce qui serait des *tables*.

La bibliothèque RTA est la colle qui lie nos tableaux ou les listes liées des structures de données aux clients PostgreSQL. L'architecture d'une application utilisant RTA devrait ressembler à quelque chose comme...



Nous l'appelons ici une *interface de gestion* car elle est destinée aux statuts, aux statistiques et à la configuration. Bien qu'une seule interface ne soit montrée, vous devriez vous rappeler que vous avez plusieurs interfaces pour votre application et elles peuvent toutes accéder à l'application simultanément.

PosgreSQL utilise TCP comme protocole de transport. Votre application doit donc être capable de se relier à un port TCP et accepter les connections des différentes interfaces utilisateur. Tous les octets reçus d'une connexion acceptée sont passés dans RTA en utilisant la sous−routine **dbcommand()**. Toutes les données devant être retournées au client sont dans un tampon retourné depuis **dbcommand()**.

Comment RTA sait−il quelles tables sont disponibles? Vous devez le lui indiquer.

Définir les tables

Vous indiquez à RTA la nature de vos tables avec la structure de données et en appelant la sous−routine **rta_add_table()**. La structure de données **TBLDEF** décrit une table et la struture **COLDEF** décrit une colonne. Vous avez ici un exemple qui illustre comment ajouter une table à l'interface RTA.

Disons que vous avez une structure de données qui possède une chaîne d'une longueur de 20 caractères et un entier, et que vous voulez exporter une table avec 5 de ces structures. Vous pouvez définir la structure et la table comme suit:

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

struct myrow mytable[5];
```

Chaque champ dans la structure de données `myrow` est une colonne dans une table d'une base de données. Nous avons besoin d'indiquer à RTA le nom de la colonne, quelle table est incluse, ses types de données, son offset (décalage) par rapport au début de la ligne et s'il est ou non en lecture seule. Nous pouvons définir des routines de *callback* qui sont appelées avant que la colonne ne soit lue et/ou après qu'elle ait été écrite. Pour notre exemple, nous supposerons que `count` est en lecture seule et que nous voulons que `do_note()` soit appelée s'il y a une écriture dans le champ `note`. Nous construisons un tableau de **COLDEF** qui est ajouté à **TBLDEF** et qui a un **COLDEF** pour chaque membre de la structure.

```
COLDEF mycols[] = {
    {
        "atable",          // table name for SQL - nom de la table
        "note",            // column name for SQL - nom de la colonne dans la table
        RTA_STR,           // data type of column/field - type des données de la colonne
        20,                // width of column in bytes - taille des données de la colonne (en
        0,                 // offset from start of row - position de la colonne
        0,                 // bitwise OR of boolean flags - masque bit-à-bit de paramètres bo
        (void (*)()) 0,    // called before read - fonction appelée avant lecture
        do_note(),         // called after write - fonction appelée après lecture
        "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."
        " - Le dernier champ d'une définition de colonne est une chaîne "
        "descriptive, l'endroit idéal pour expliciter le format des "
        "données et la façon de les utiliser."
    },
    {
        "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."
            " - Si vos tables réalisent l'interface entre l'(les )interface(s) "
            "utilisateur et le service, alors les commentaires dans la "
            "définition des tables et colonnes constituent la "
            "spécification fonctionnelle de votre projet et peuvent "
            "se révéler la meilleure source de documentation "
            "pour les développeurs."
    };
```

L'écriture des callbacks peut être le système de pilotage réel de votre application. Vous pouvez vouloir des changements dans le déclenchement d'une table pour d'autres changements ou une reconfiguration de votre application.

Vous indiquez les tables à RTA en lui donnant le nom de la table, la longueur de chaque ligne, un tableau de COLDEFS pour décrire les colonnes, un compte des colonnes, le nom du fichier sauvegardé si vous voulez que quelques unes des colonnes soient non volatiles et une chaîne pour décrire la table. Si la table est un tableau statique de struct, vous donnez l'adresse de départ et le nombre de lignes dans la table. Si la table est implementée comme une liste liée, vous donnez une routine à RTA pour appeler une routine qui fait une *itération* d'une ligne vers la suivante.

```
TBLDEF   mytableDef = {
      "atable",                 // table name - nom de la table
      mytable,                  // address of table - adresse de la table
      sizeof(myrow),            // length of each row - taille d'une ligne (ou entrée)
      5,                        // number of rows - nombre de lignes
      (void *) NULL,            // iterator function - fonction d'itération
      (void *) NULL,            // iterator callback data -itérateur de données de callback
      mycols,                   // Column definitions - définitions des colonnes
      sizeof(mycols / sizeof(COLDEF), // # columns - colonnes
      "",                       // save file name - fichier de sauvegarde
      "A complete description of the table."
      " - Description complète de la table."
};
```

Vous voudrez normalement que le nom de la table vue par SQL soit le même que celui du programme. L'exemple est pris de `mytable` vers `atable` juste pour montrer que les noms n'ont pas besoin d'être identiques.

Avec tout le code donné ci–dessus, vous pouvez maintenant donner des informations à RTA à propos de votre table.

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

Tout est ici pour le réaliser. Pour utiliser RTA, vous avez besoin d'apprendre comment utiliser les deux structures de données (COLDEF et TBLDEF), deux sous–routines (dbcommand() et rta_add_table()).

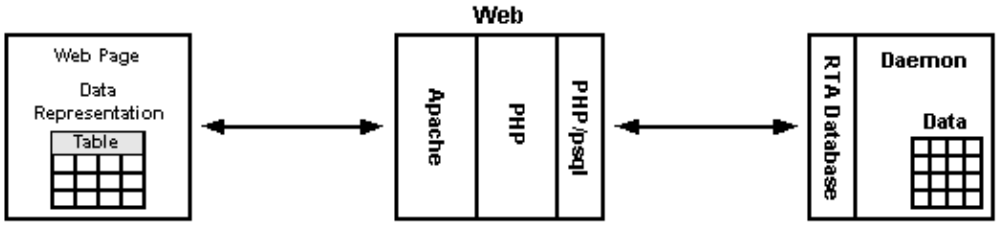
Le code ci–dessus est destiné à vous donner un avant–goût du fonctionnement de RTA. Il n'est pas destiné à être un tutoriel complet ou un exemple de travail complet. Un exemple de travail complet et une description complète de l'API RTA et des structures de données peuvent être trouvées sur le site web de RTA (www.runtimeaccess.com).

Vous devez simplement définir les tables à utiliser pour l'utilisation dans votre application, RTA définit donc son propre jeu de tables internes. Les deux tables les plus intéressantes sont rta_tables et rta_columns qui

forment, bien sûr, les tables des listes et la description de toutes les tables et colonnes que vous avez définies. Ce sont les fameuses *tables système*. Les tables système sont pour une base de données ce que `ls` est pour un système de fichiers et que `getnext()` est pour SNMP.

L'éditeur de table

Un des utilitaires qui est fourni avec RTA est un petit programme PHP qui utilise les tables système pour lister vos tables RTA dans une fenêtre de navigateur web. Les noms des tables sont liés et cliquer sur un nom de table affiche les 20 premières lignes de la table. Si la table possède plusieurs champs éditables, vous pouvez cliquer sur une ligne pour ouvrir une fenêtre d'édition pour cette ligne. Tout ceci est réalisé en utilisant les descriptions de table et de colonne trouvées dans les tables système. Le flux de données est décrit dans le diagramme suivant.



La vue de la synthèse de l'éditeur de table pour l'application d'exemple de RTA est montrée ci−dessous.

Editeur de table RTA

Nom de la table	Description
rta_tables	La table de toutes les tables dans le système. C'est une pseudo table et non un tableau de structures comme les autres tables.
rta_columns	La liste de toutes les colonnes dans toutes les tables avec leurs attributs.
pg_user	La table des utilisateurs Postgres. Nous modifions cette table de telle manière que tout nom d'utilisateur dans une clause WHERE apparaît dans la table comme utilisateur légitime sans être super utilisateur, sans createDB, sans trace ou catupd.
rta_dbg	Configuration du journal de débogage. Un callback sur le champ 'target' ferme et réouvre syslog(). Aucune des valeurs de cette table n'est sauvegardée sur le disque. Si vous souhaitez des valeurs par défaut, vous devez changer le source RTA ou effectuer un SQL_string() pour initialiser les valeurs lorsque vous initialisez votre programme.
rta_stat	Utilisation et compte des erreurs pour le paquetage RTA.

mytable	Un exemple de table d'application
UIConns	Données relatives aux connexions TCP pour les programmes d'interface utilisateur

Au fait, si tout s'est bien passé dans la publication de cet article de LinuxFocus, les noms de tables donnés ci−dessus doivent avoir des liaisons avec l'application exemple fonctionnant sur le serveur web RTA de Santa Clara, California. Un bon lien à suivre est `mytable`...

Deux commandes

Run Time Access est une bibliothèque qui lie les programmes de gestion ou d'interface utilisateur écrits avec la bibliothèque client PostgreSQL (libpq) à votre application ou démon. RTA est une interface, pas une base de données. Comme telle, elle ne nécessite que deux commandes SQL, SELECT et UPDATE.

La syntaxe pour la déclaration SELECT est:

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

La column_list est un liste de noms de colonnes séparés par des virgules. La where_clause et une liste de comparaisons séparées par des AND (et). Les opérateurs des comparaison sont =, |=, >=, <=, >, et <. Une limit_clause a la forme `[LIMIT i] [OFFSET j]`, où `i` est le nombre maximum de lignes à retourner et où nous sautons les `j` premières lignes avant de démarrer la sortie. Quelques exemples peuvent aider à clarifier la syntaxe.

```
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
```

Positionner LIMIT à 1 et spécifier un OFFSET est une manière d'avoir une ligne spécifique. Le dernier exemple ci−dessus est équivalent au code C `(mytable[3].count)`.

La syntaxe de la déclaration UPDATE est:

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

Les clauses where_clause et limit sont décrites au−dessus. update_list est une liste d'assignations de colonnes (nouvelles valeurs) séparées par des virgules. Nous nous aiderons avec quelques exemples.

```
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 reconnaît les mots réservés en majuscules et les minuscules bien que les exemples ici soient en majuscules pour tous les mots réservés SQL.

Téléchargement et Construction

Vous pouvez télécharger RTA depuis ce site web sur www.runtimeaccess.com (le copyright de RTA est en LGPL). Faites attention dans la sélection des versions de RTA pour le téléchargement. La dernière version RTA utilise le nouveau protocole PostgreSQL introduit avec la version 7.4 de PostgreSQL. La plupart des distributions Linux utilisent la version 7.3. Lorsque vous utilisez une version plus ancienne de RTA pour les essais initiaux, vous devriez utiliser la dernière version pour avoir les derniers correctifs de bogues et les améliorations.

Décompacter le paquet vous donnera les répertoires suivants:

```
./doc            # copie du site Web RTA
./empd           # prototype de démon construit avec RTA
./src            # fichiers sources de la bibliothèque RTA
./table_editor   # sources PHP de l'éditeur de tables
./test           # code source d'une application d'exemple
./util           # utilitaires RTA
```

Grâce à Graham Phillips, la version 1.0 de RTA possède un support autoconf. Graham a porté RTA depuis Linux vers Mac OS X, Windows et FreeBSD. En utilisant la version 1.0, vous pouvez construire RTA avec l'habituel

```
./configure
make
make install      # (en tant que root)
```

L'installation met librtadb.so et les fichiers bibliothèques associés dans le répertoire /usr/local/lib. Pour utiliser RTA, vous pouvez ajouter ce répertoire dans /etc/ld.so.conf et lancez la commande ldconfig ou vous pouvez ajouter le répertoire dans votre chemin de chargement avec:

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

L'installation met le fichier d'en–tête de RTA, rta.h, dans /usr/local/include.

make construit un programme de test dans le répertoire `test` et vous pouvez tester votre installation en changeant de répertoire vers le répertoire de test et en lançant `./app &`. Un `netstat -nat` devrait montrer un programme écoutant sur le port 8888. Vous pouvez maintenant lancer psql et effectuer des commandes SQL sur votre application test.

```
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)
```

Alors qu'il semble que vous soyez connecté à une base de données, vous ne l'êtes pas. N'oubliez pas: les seules deux commandes que vous pouvez utiliser sont SELECT et UPDATE.

Les avantages de RTA

Les avantages de séparer les programmes de l'interface utilisateur du démon tombent dans les grandes catégories de la conception, du codage, du déboguage et des aptitudes.

Du point de vue de la conception, la division vous force à décider tôt dans la conception ce qui est exactement offert comme partie de l'interface utilisateur sans vous préoccuper de comment il est affiché. Le processus de pensée nécessaire pour concevoir les tables vous force à penser à la conception réelle de votre application. Les tables peuvent former la spécification fonctionnelle interne de votre application.

Lors du codage, les définitions de tables sont en fonction de ce que les ingénieurs du démon ont développé et comment les ingénieurs de l'interface utilisateurs en ont tiré parti. La division de l'interface utilisateur et du démon signifie que vous pouvez engager des experts d'interfaces graphiques et des experts de démons indépendemment et ils peuvent coder isolément, ce qui peut aider à mettre le produit sur le marché plus rapidement. Comme il existe des liaisons pour PHP, Tcl/Tk, Perl et "C", vos développeurs peuvent utiliser les bons outils pour leur travail.

Le déboguage est plus rapide et plus facile car l'interface utilisateur et le démon peuvent simuler l'autre partie sans problème. Par exemple, les ingénieurs d'interfaces utilisateur peuvent faire fonctionner leurs programmes avec une base de données Postgres avec les mêmes tables qu'utilise le démon. Tester le démon peut être facile et plus complet car il est plus facile de construire les scripts de test pour simuler les interfaces utilisateurs et il est plus facile d'examiner les statuts et les statistiques internes lors des essais. L'aptitude à forcer un état interne ou des conditions aide aux tests des effets de bord, ce qui est parfois difficile Ã réaliser dans des conditions de laboratoire.

La capacité de votre produit peut être étendue avec RTA. Vos clients apprécieront réellement de voir les informations de statut et de statistiques lorsque le programme est lancé. La séparation de l'interface du démon signifie que vous pouvez avoir plus de programmes d'interfaces: SNMP, ligne de commande, web, LDAP, et la liste n'est pas limitative... Cette flexibilité sera importante pour vous si (alors!) vos clients demandent des interfaces personnalisées.

RTA offre plusieurs autres fonctionnalités que vous pouvez souhaiter dans un paquet de ce type:

- Le modèle de données de l'application reflété par le modèle de données de l'API
- L'accès déporté aux applications
- L'utilisation de standards et de logiciels existants pour l'application
- Peu de nouveaux protocoles et d'API à apprendre
- Des mécanismes de découverte pour l'application
- Peu de contraintes sur l'application
- Un verrouillage des ressources
- Une efficacité du CPU et de la mémoire

Résumé

Cet article a présenté une très brève introduction à la bibliothèque RTA et ses possibilités. Le site web RTA possède une FAQ, une description complète de l'API et plusieurs programmes clients d'exemple.

Comme RTA peut rendre vos structures de données visibles comme des tables dans une base de données, il peut donc les rendre visibles comme des fichiers dans un système de fichiers virtuel. (En utilisant le paquet File System dans Userspace (FUSE) par Miklos Szeredi.) Le site web possède plus d'informations sur comment utiliser l'interface du système de fichiers.
