NAME

DBD::Proxy - A proxy driver for the DBI

SYNOPSIS

```
use DBI;

$dbh = DBI->connect("dbi:Proxy:hostname=$host;port=$port;dsn=$db",
$user, $passwd);

# See the DBI module documentation for full details
```

DESCRIPTION

DBD::Proxy is a Perl module for connecting to a database via a remote DBI driver. See DBD::Gofer for an alternative with different trade-offs.

This is of course not needed for DBI drivers which already support connecting to a remote database, but there are engines which don't offer network connectivity.

Another application is offering database access through a firewall, as the driver offers query based restrictions. For example you can restrict queries to exactly those that are used in a given CGI application.

Speaking of CGI, another application is (or rather, will be) to reduce the database connect/disconnect overhead from CGI scripts by using proxying the connect_cached method. The proxy server will hold the database connections open in a cache. The CGI script then trades the database connect/disconnect overhead for the DBD::Proxy connect/disconnect overhead which is typically much less.

CONNECTING TO THE DATABASE

Before connecting to a remote database, you must ensure, that a Proxy server is running on the remote machine. There's no default port, so you have to ask your system administrator for the port number. See DBI::ProxyServer for details.

Say, your Proxy server is running on machine ''alpha'', port 3334, and you'd like to connect to an ODBC database called ''mydb'' as user ''joe'' with password ''hello''. When using DBD::ODBC directly, you'd do a

```
$dbh = DBI->connect("DBI:ODBC:mydb", "joe", "hello");
```

With DBD::Proxy this becomes

```
$dsn = "DBI:Proxy:hostname=alpha;port=3334;dsn=DBI:ODBC:mydb";
$dbh = DBI->connect($dsn, "joe", "hello");
```

You see, this is mainly the same. The DBD::Proxy module will create a connection to the Proxy server on ''alpha'' which in turn will connect to the ODBC database.

Refer to the DBI documentation on the `connect` method for a way to automatically use DBD::Proxy without having to change your code.

DBD::Proxy's DSN string has the format

```
$dsn = "DBI:Proxy:key1=val1; ... ;keyN=valN;dsn=valDSN";
```

In other words, it is a collection of key/value pairs. The following keys are recognized:

hostname
port

Hostname and port of the Proxy server; these keys must be present, no defaults. Example:

```
hostname=alpha;port=3334
```

dsn

The value of this attribute will be used as a dsn name by the Proxy server. Thus it must have the format `DBI:driver:...`, in particular it will contain colons. The *dsn* value may

contain semicolons, hence this key *must* be the last and it's value will be the complete remaining part of the dsn. Example:

```
dsn=DBI:ODBC:mydb
```

cipher
key
usercipher
userkey
By using these fields you can enable encryption. If you set, for example,

```
cipher=$class;key=$key
```

(note the semicolon) then DBD::Proxy will create a new cipher object by executing

```
$cipherRef = $class->new(pack("H*", $key));
```

and pass this object to the RPC::PlClient module when creating a client. See RPC::PlClient. Example:

```
cipher=IDEA;key=97cd2375efa329aceef2098babdc9721
```

The usercipher/userkey attributes allow you to use two phase encryption: The cipher/key encryption will be used in the login and authorisation phase. Once the client is authorised, he will change to usercipher/userkey encryption. Thus the cipher/key pair is a **host** based secret, typically less secure than the usercipher/userkey secret and readable by anyone. The usercipher/userkey secret is **your** private secret.

Of course encryption requires an appropriately configured server. See ''CONFIGURATION FILE'' in DBD::ProxyServer.

debug
Turn on debugging mode

stderr
This attribute will set the corresponding attribute of the RPC::PlClient object, thus logging will not use *syslog()*, but redirected to stderr. This is the default under Windows.

```
stderr=1
```

logfile
Similar to the stderr attribute, but output will be redirected to the given file.

```
logfile=/dev/null
```

RowCacheSize
The DBD::Proxy driver supports this attribute (which is DBI standard, as of DBI 1.02). It's used to reduce network round-trips by fetching multiple rows in one go. The current default value is 20, but this may change.

proxy_no_finish
This attribute can be used to reduce network traffic: If the application is calling `$sth->`*finish()* then the proxy tells the server to finish the remote statement handle. Of course this slows down things quite a lot, but is perfectly good for reducing memory usage with persistent connections.

However, if you set the *proxy_no_finish* attribute to a TRUE value, either in the database handle or in the statement handle, then *finish()* calls will be suppressed. This is what you want, for example, in small and fast CGI applications.

proxy_quote
This attribute can be used to reduce network traffic: By default calls to `$dbh->`*quote()* are passed to the remote driver. Of course this slows down things quite a lot, but is the safest default behaviour.

However, if you set the *proxy_quote* attribute to the value '`local`' either in the database handle or in the statement handle, and the call to quote has only one parameter, then the local default DBI quote method will be used (which will be faster but may be wrong).

KNOWN ISSUES

Unproxied method calls

If a method isn't being proxied, try declaring a stub sub in the appropriate package (DBD::Proxy::db for a dbh method, and DBD::Proxy::st for an sth method). For example:

```
sub DBD::Proxy::db::selectall_arrayref;
```

That will enable selectall_arrayref to be proxied.

Currently many methods aren't explicitly proxied and so you get the DBI's default methods executed on the client.

Some of those methods, like selectall_arrayref, may then call other methods that are proxied (selectall_arrayref calls fetchall_arrayref which calls fetch which is proxied). So things may appear to work but operate more slowly than the could.

This may all change in a later version.

Complex handle attributes

Sometimes handles are having complex attributes like hash refs or array refs and not simple strings or integers. For example, with DBD::CSV, you would like to write something like

```
$dbh->{"csv_tables"}->{"passwd"} =
{ "sep_char" => ":", "eol" => "\n";
```

The above example would advice the CSV driver to assume the file ''passwd'' to be in the format of the /etc/passwd file: Colons as separators and a line feed without carriage return as line terminator.

Surprisingly this example doesn't work with the proxy driver. To understand the reasons, you should consider the following: The Perl compiler is executing the above example in two steps:

1. The first step is fetching the value of the key ''csv_tables'' in the handle `$dbh`. The value returned is complex, a hash ref.
2. The second step is storing some value (the right hand side of the assignment) as the key ''passwd'' in the hash ref from step 1.

This becomes a little bit clearer, if we rewrite the above code:

```
$tables = $dbh->{"csv_tables"};
$tables->{"passwd"} = { "sep_char" => ":", "eol" => "\n";
```

While the examples work fine without the proxy, the fail due to a subtle difference in step 1: By DBI magic, the hash ref `$dbh->{'csv_tables'}` is returned from the server to the client. The client creates a local copy. This local copy is the result of step 1. In other words, step 2 modifies a local copy of the hash ref, but not the server's hash ref.

The workaround is storing the modified local copy back to the server:

```
$tables = $dbh->{"csv_tables"};
$tables->{"passwd"} = { "sep_char" => ":", "eol" => "\n";
$dbh->{"csv_tables"} = $tables;
```

SECURITY WARNING

RPC::PlClient used underneath is not secure due to serializing and deserializing data with Storable module. Use the proxy driver only in trusted environment.

AUTHOR AND COPYRIGHT

Jochen Wiedmann
Am Eisteich 9
72555 Metzingen
Germany

Email: joe@ispsoft.de
Phone: +49 7123 14887

SEE ALSO

DBI, RPC::PlClient, Storable