Database proxies in Perl

Bernd Ulmann
ulmann@vaxman.de

IT-Symposium 2007
16th April – 20th April 2007
Nuremberg

Introduction

- About two years ago I ported Germany's largest vegan recipe database from a LINUX system running MySQL and Apache to an OpenVMS system (a VAX-7820) running RDB and the WASD web server.
- During this port several severe problems were encountered which eventually led to the development of a database proxy written in Perl which not only solved these problems but proved to be a very useful tool in a variety of other situations as well[1].
- Those initial problems which led to the decision to use a three tier architecture involving a proxy were:

[1] I would like to thank my friend Thomas Kratz who did most of the work described in the following.

Intodution

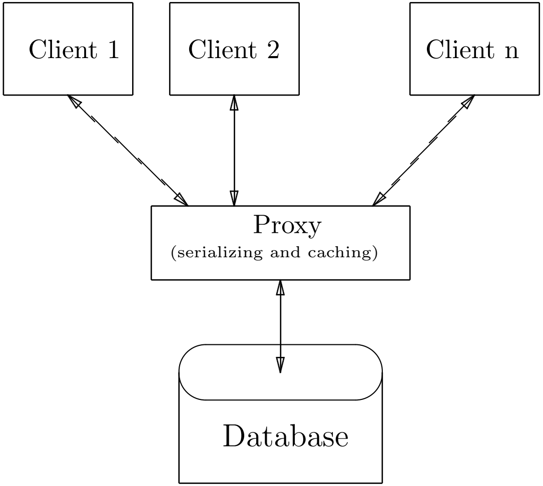
1. The initial overall response time of the system comprised of RDB, several CGI-scripts written in Perl and the WASD server was poor on the VAX.
   The response times got somewhat better by using the RTE feature of the WAS server thus allowing a memory resident Perl interpreter (this eventually led to a performance gain of 3 to 5).
2. Concurrent accesses sometimes led to mysterious crashes in the database interface module used to access RDB from within a Perl program.
   It turned out that this problem was buried deep in the interface routines and could not be solved easily. Crashes only occured during simultaneous connects to the database – serialized connects always worked fine.

Employing a proxy

- The frequent crashes forced an access method which allowed to serialize database connects and accesses in order to prevent these crashes.
- Something like this can be done easily using a so called *Proxy*:
  "A proxy server is a computer that offers a computer . . . service to allow clients to make indirect . . . connection to other . . . services."[2]
- What was needed was a proxy which accepts connections and request from the CGI scripts forming the web interface of the recipe database. These requests will be serialized and routed to the central database of the system. In addition to this, the proxy might cache results read from the database so many requests may be handled without any database access at all after some startup time.

[2]Source: Wikipedia

System overview



Why Perl?

It was decided to use the programming language Perl to implement the proxy due to a couple of reasons:

1. Thomas and I really like Perl!
2. Perl is a mighty programming language with very efficient methods for manipulating data and there are countless modules simplifying database accesses, TCPIP-communication, etc.
3. The availability of a hash as a basic datastructure makes the implementation of caching mechanisms very simple.
4. Perl is readily available for nearly every platform, including OpenVMS.

Client/proxy communication

The client and proxy communicate by transmitting complete Perl datastructures (hashes) using sockets. This has the advantage that no impedance mismatch between client and server occurs and manipulating requests and data can be done in a transparent and natural way.
A typical request sent from a CGI-script to the proxy may look like this (it is a hash with three keys):

Example request

```
{
  SELECT => 'SELECT DESCRIPTION FROM RECIPIES',
  ALIAS => 'RECIPES_PRODUCTION',
  KEEP => 1,
}
```

Structure of a request

A request like this consists of three parts:

1. The SQL-statement to be executed or satisfied by a cache access. The key of the hash element containing the actual statement determines its type (`SELECT`, `UPDATE`, `DELETE`).
2. An alias pointing to the desired database – this allows a single proxy server to hold several connections to different databases at once while every client selects the database it wants to query.
3. An optional `KEEP`-clause which controls the cache invalidation mechanism as will be described later.

Transmitting a request

To transmit such a request to the proxy the client program makes use of the `freeze` function from the `Storable` module to pack the hash containing the request and send it to a socket in base-64 encoded format:

Transmitting a request

```
print $sock encode_base64(nfreeze(
    {
        $stmt_type => $stmt,
        ALIAS => 'RECIPES_PRODUCTION',
    }
), '', ), "\n";
```

Receiving a request

The server reads this base-64 data stream from the socket and recreates the datastructure like this:

Receiving a request

```
my $request_ref = eval {thaw(decode_base64($raw))};
```

`$request_ref` is now a reference to an exact copy of the datastructure sent by the client and may be used in the following steps in a native way.

Transmitting the result data set

Sending the data set resulting from a client's request is a bit more tricky than sending the request:

- This is due to the fact that a result data set may easily exceed several 100 kB in size which can not be sent through a socket in a single step.
- Therefore it is necessary to split the data set to be transmitted into smaller chunks and send these to the client which, in turn, has to reassemble these chunks into the final result data set.

Transmitting the result data set

Assuming that $response is a scalar containing the data to be sent from the proxy to the client, while $chunk_size contains the maximum size of chunks, the transmission is done like this ($eod holds some character sequence denoting the end of a transmission):

Transmitting the result data set

```
print $sock $_, "\n"
    for unpack("A$chunk_size" x (int(length($response) /
        $chunk_size) + 1), $response);

print $sock $eod, "\n";
```

Receiving the result data set

Receiving the data set sent from the proxy to the client is easy:

Receiving the result data set

```
my $buf;
do {
    $buf .= <$sock>;
} until $buf =~ s/$eod$//m;
my $response = eval {thaw(decode_base64($buf))};
```

Operation of the proxy

The main part of the proxy is an endless loop waiting for incoming packets on the socket of the proxy.
Whenever a request is received by the proxy the following actions occur:

- If it is a `SELECT`-statement, the whole statement will be used as a key to a cashing hash. If there is a corresponding entry in the hash, the proxy will return this entry without actually querying the database.
- A `DELETE`-statement will invalidate all entries in the cache having keys containing references to one or more of the tables specified in the `DELETE`-statement.

Operation of the proxy

- An UPDATE-statement normally has the same effect as a DELETE-statement, i.e. it will flush all possibly affected cache entries.
  Since there are situations where such a behaviour is undesirable, it is possible to influence this mechanism by specifying an additional entry in the request hash sent from the client to the proxy. If there is an entry KEEP with an associated true value, the corresponding UPDATE-statement will be performed without flushing any cache entries at all. This is a very useful feature in cases where, for example, only some counters will be incremented without affecting any vital information in the database.

Operation of the proxy

Since only UPDATE- and DELETE-statements will force the proxy to clear any cache entries, there is some danger of hitting the allowed maximum memory size for a process when performing mostly SELECT-statements.
To avoid this, the proxy will scan its cache in regular (configurable) intervals for entries which have not been accessed for a longer period than some (also configurable) threshold.
Using this mechanism the proxy will only hold entries in its cache which are quite likely to be used while seldom used data will be flushed after a while.

Experiences

The speedup effect resulting from using this proxy is dramatic: A typical request to generate the category selection list for the recipe database web application takes about 3.2 seconds using RDB without the proxy.
Using the proxy the very same request needs 3.52 seconds to complete – every following request of this type takes only 0.34 seconds.
The same speedup by a factor of ten can be observed with other requests.
The following slide shows some process information about the proxy after a run time of nearly 100 days:

Experiences

SH PROC

```
18-MAR-2007 User:  HTTP$NOBODY Process ID: 20223090
Node:  FAFNER Process name:  "BATCH_743"

Accounting information:
Buffered I/O count:      45803428
Peak working set size:   131072
Direct I/O count:        206717
Peak virtual size:       213000
Page faults:             147660
Elapsed CPU time:        0 07:05:57.47
Connect time:            94 14:58:22.41
```

Additional benefits

Using a proxy process as the central hub in a multi user database application brings some additional benefits which have not been mentioned already:

- The central position of the proxy allows the efficient implementation of security relevant checks. A prime example for this is the increasing amount of SQL-injections encountered in many database applications offering a web interface. Tainted SQL requests like those resulting from injection attempts may be filtered out easily using the proxy.

Additional benefits

- In addition to this, the central role of the proxy makes the generation and extraction of detailed statistical data quite easy. A proxy like the one described above may generate statistical data on the fly, giving near realtime information about the usage of complex database applications.
- Since the proxy can hold multiple connections to different databases it can be used to harmonize the interface visible to the client processes. This became most obvious while using a mixture of MySQL and RDB for which at least the connection methods differ substantially.

Fazit

The use of a (still simple) database proxy like the one described above has shown to be very fruitful:

1. On the system used, a VAX-7820, an overall speedup by a factor of 10 was observed.
2. The price for this was some CPU overhead which was more than compensated by the decreased amount of database requests actually being performed. Only the memory consumption of the proxy has to be taken into account – a working set smaller than 131072 pages has shown to be suboptimal.
3. The harmonized call interface for various databases has proven to be a valuable tool since it allows easy migration between different databases.
4. The possibility of security checks in the proxy will be explored in the near future due to an increasing amount of SQL injection attempts from malicious internet users.

Additional information

- The vegan recipe database which initially triggered the development of the database proxy described in this talk is the heart of
  `http://www.veganwelt.de`
  and may be reached directly at
  `http://fafner.dyndns.org/∼mitsam/cgi-bin/rz1.pl`
- If you have questions, comments or would like to receive a copy of the proxy server, the author may be reached at
  `ulmann@vaxman.de`