Perl and OpenVMS

Bernd Ulmann
ulmann@vaxman.de

IT-Symposium 2008
4th June – 5th June 2008
Frankfurt

Outline

1 Introduction

2 VMS specialities

3 Examples

4 Conclusion and contact

Overview

On the following slides I try to give an impression of the power of Perl in general and in special in an OpenVMS environment.

The examples were chosen from my own daily work on a large OpenVMS system and range from simple code snippets, programs for one time usage to larger Perl programs running as batch jobs and performing crucial tasks like fetching mail from a POP-server and the like.

Perl is no replacement for DCL but it makes life much more easier when it comes to file parsing and modifying, socket communication, data base accesses etc.

Some facts about Perl

The following slides contain some (very) basic facts about Perl which are in no way complete but may give an impression of the programming language for those who have never seen any actual Perl code.

For Perl programmers there exists one main resource of information and enlightenment – the so called *camel book*, "Programming Perl", published by O'Reilly.

What is Perl?

- Perl was developed by Larry Wall.
- The name is not an acrynom but a retronym.
- Perl is a modern interpreter language (in fact the interpreter does a good job precompiling the code so Perl programs tend to run surprisingly fast).
- Perl runs on more architectures and operating systems than most other languages (including Java).
- Perl code looks strange for the novice.
- Perl is incredibly powerful and concise.
- Much of the power of Perl lies in its modules.
- Perl's philosophy is "*There is more than one way to do it.*"

Variables

- Perl does not care about the type of a variable, it is type free.
- Instead, Perl cares about the structure of a variable – in essence there are three basic structures:

  Scalars: A scalar variable can hold a single value at a time and the variable name is always preceeded by a `$` like in `my $pi = 3.14159265;`

  Arrays: An array is an indexed list consisting of scalars as its elements. The name of an array is preceeded by a `@` like in `my @entries;`

  Hashes: These are similar to arrays but use strings instead of numerical indices. Their names are preceeded by a `%` like in `my %data;`

Variables

Some examples of Variables

```
my $pi = 'three_point_one_four';

my @array;
$array[0] = 'Something';
$array[1] = 'Something else';

my %data;
$data{'name'} = 'Bernd';
$data{'occupation'}= 'VAX enthusiast';
```

Control structures

Perl supports all of the common control structures like `if ...else`, `while`, `do ...while`, `for` etc. Every such statement requires a block surrounded by braces – these can not be left out like in C if only a single statement is to be controlled. In this special case so called *statement modifiers* may be used:

Some examples of control structures

```
if (condition)                  # traditional if
{
  ...
}

statement if condition          # statement modifier;
```

Control structures

Some examples of control structures

```
for (my $i = 0; $i < 10; $i++)
{
  print "$i\n";
}

for my $value (@array)
{
  print "$value\n";
}

print "$_\n" for (@array);
```

Regular expressions

Much of Perl's power stems from its built in regular expression parser. Regular expressions tend to look a bit like line noise so I will not give a reasonable example here – you will see some in the examples below.

Modules

Another source of Perl's power is the multitude of modules which are available on CPAN.

Regardless what your initial problem is, first have a look for a module which might be helpful for your task.

Examples for the power of modules are numerous – in the following slides a program to fetch mails from a POP3-server is described which uses a POP3-module for example. This module contains all necessary functionality to connect to a POP3-server and get mails, delete mails etc. With the help of the `Net::POP3`-Module these tasks reduce to simple calls.

Do/do not

Do:

- Be open for the Perl way to solve problems. Perl programs tend to be quite short and powerful.
- Expect code that is significantly shorter than equivalent code in other languages like C etc.
- Use statements like `split`, `join`, `map`, `grep` instead of unnecessary loops.
- Use regular expressions to parse, manipulate or split strings!
- Be strict and use warnings all the time!
- Get the *Camel Book*!

Do not:

- . . . program in Perl like in C etc.
- . . . use arrays when you can use hashes.
- . . . loop over an array to find an element!

Resources

The central repository for Perl and the thousands of modules available is

`http://www.cpan.org`

This is the *C*omprehensive *P*erl *A*rchive *N*etwork.

Whenever you need a Perl interpreter, a module or documentation, have a look at the CPAN web site or search directly via

`http://search.cpan.org`

Installing Perl

Basically there are two ways to get a Perl interpreter up and running on a system:

1. You can use a precompiled package – for OpenVMS there is a HP supplied distribution kit, for other platforms there are numerous such kits available.
2. Get the sources and compile and install the system yourself.

Personally, I would always prefer the second method since I like to know what is really running on my system and sometimes I want to do some things differently compared with a precompiled installation kit. (Compiling a Perl system on an Alpha is fast, but on a VAX this can take several hours, be prepared!)

VMS modules

As all of you know, OpenVMS is different from (and superior to :-) other operating systems which has to be taken into account when porting or writing software.

There are a lot of modules encapsuling OpenVMS specific tasks like interfacing the mail system and the like as well as there are modules which take into account that there is a variety of different ways to handle file and directory names etc.

In the following a selection of modules which I found to be especially useful in an OpenVMS environment is listed:

Some useful modules for OpenVMS systems

`VMS::Device`	Interfache to `$GETDVI` and the like.
`VMS::Filespec`	Converts between OpenVMS and UNIX file specs.
`VMS::FlatFile`	Use hashes to work with index files.
`VMS::ICC`	Intra cluster communication services.
`VMS::Mail`	Interface to the OpenVMS mail system.
`VMS::Process`	Manage OpenVMS processes.
`VMS::Queue`	Work with queues and their entries.
`VMS::Stdio`	File operations like binmode, flush, vmsopen etc.
`VMS::System`	Retrieve system information.
`File::Basename`	System independent operations on filenames etc.

Examples

The following slides show some practical examples of using Perl in a productive OpenVMS environment.

Some examples, especially the simpler ones, are accompanied with their source code which might be interesting, although more complex examples are only described.

If you are interested in the source code of one of these more complex examples, please let me know, I will make it available to you by mail upon request.

Programs for one time usage

Many everyday tasks require that system administrators as well as programmers perform some unexpected tasks like clever pattern matching, parsing log files and the like which are not readily solved with standard DCL tools.

Many of these problems can be solved on the fly using a couple of lines of Perl code.

The following sections show some typical examples from my own everyday work using OpenVMS.

Perl as a command line tool

Perl can be used as a command line tool like, for example, `awk` on UNIX systems. This can be very useful when you have a puzzling problem which does not deserve a real program but needs a clever data conversion on the fly or something like that.

Since there are a variety of command line options for Perl which are useful in this context, only a simple example is given in the following. More information about this topic may be found elsewhere like the Camel book etc.

Changing a configuration file

Once I inherited a configuration file `TRANSFER.INI` which looked in parts like this:

```
[logging]
  log    = log/transfer.log
  ticket = log/ticket.log
[templates]
  ticket = templates/ticket.tpl
  mail   = templates/mail.tpl
```

Of course these path names are not very OpenVMS like and it would have been quite cumbersome to edit all of them (there was a lot of those path names) by hand.

Changing a configuration file

One could write a small Perl program reading the file, performing the necessary changes using regular expressions and writing the result back to disk.

Since tasks like these are commonplace, Perl can be used as a mighty command line tool for performing in place edit operations like transforming these path names into OpenVMS file names:

In place editing:

```
perl -i -pe -
"s/^(.*\s*)=(\s*)(.+)\/(.+)/$1=$2\[\.$3\]$4/" i-
transfer.ini
```

Changing a configuration file

Applying this single line statement to the configuration file shown above, the resulting file looks like this:

```
[logging]
  log    = [.log]transfer.log
  ticket = [.log]ticket.log
[templates]
  ticket = [.templates]ticket.tpl
  mail   = [.templates]mail.tpl
```

Finding unresolved bibliography items

When writing LaTeX-documents including a bibliography without using BibTeX, there is some risk of having uncited bibliography entries in the source code. A bibliography entry has the form

```
\bibitem{zachary} %book
G. Pascal Zachary, \emph{Endless Frontier --
Vannevar Bush, Engineer of the American Century},
The MIT Press, 1999
```

While a citation looks like

```
cf.  \cite{zachary}[p.  142].
```

Finding unresolved bibliography items

Having a document of more than 120000 lines resulting in about 600 pages of text with more than 600 bibligraphy entries, I needed a way to be sure that every entry was in fact cited in the text.

To accomplish this I wrote the following short Perl program which reads in the complete source code with a single statement and parses this for all citations in a first run which builds a hash containing all citations, followed by a second run looking for all bibliography entries.

Entries without corresponding citation will be printed to stdout (it turned out that more than 20 entries in the text were unused).

Finding uncited entries in LaTeX-source

```
use strict;
use warnings;

die "Usage bib.pl <filename.tex>\n" unless @ARGV + 0;

my $data;
open my $fh, '<', $ARGV[0] or die "Could not open $ARGV[0]:  $!\n";
{
  local $/;
  $data = <$fh>;
}
close $fh;

my %cite;
$cite{$_}++ for $data =~ m/\\cite\{(.+?)\}/g;


$cite{$_} or print "$_\n" for $data =~ m/\\bibitem\{(.+?)\}/g;
```

Parsing a log file

Some weeks ago I had to parse a log file with entries like shown in the following for some timestamps to calculate an average time value.

```
[LOG|SYSTEM|2008 May 13, 14:15:26 (886)|ENGINE.batch]
Loaded 16 events in 497 milliSecs
[END]
[LOG|SYSTEM|2008 May 13, 14:15:55 (281)|Risk|BatchJob]
Time to execute Scenario 24902 ms
[END]
[LOG|SYSTEM|2008 May 13, 14:15:55 (283)|RiskAnalysis|BatchJob]
ScenarioAnalysis Executed [SUCCESS]
[END]
[LOG|SYSTEM|2008 May 13, 14:16:12 (870)|Risk|BatchJobThread]
Time to execute Scenario 13662 ms
[END]
```

Parsing a log file

Since these logfiles grow quite fast and since it is impossible to restart logging into a new file when a new round of tests is performed, the calculation of average times must be possible from any point in the file on starting with a given date and timestamp.

Using a simple `SEARCH` is not too easy since log entries are at least three lines in length and since I am only interested in entries of the form `Time to execute Scenario...` after a given timestamp.

All of this called for a short Perl program to parse the file and compute the desired average execution time value.

Parsing a log file

```
use strict; use warnings;
die "Usage:  stat3 \"yyyy mmm dd\" \"hh:mm:ss\"\n" if @ARGV != 3;

my ($file, $date, $min_time) = @ARGV;
my @values;
open my $fh, '<', $file or die "Could not open $file:  $!\n";
{
  local $/ = '[END]';
  while (my $entry = <$fh>)
  {
    my ($time, $duration) = $entry =~
    m/^.+\|.+\|$date,\s(\d\d:\d\d:\d\d\s).*execute Scenario\s(\d+)\sms/s;
    push(@values, $duration) if $time and $time ge $min_time;
  }
}
close $fh;

print 'Average:  ', int(eval(join('+', @values)) / @values) / 1e3,
  ' s (', @values + 0, ")\n" if @values + 0;
```

Are there any files with W:WD on my disk?

One day I was asked "How can you be sure there are no files on your system disk which are writable by WORLD?"

Good question – this calls for a short Perl program.

Search for files with W:WD

```
use strict;
use warnings;

my ($fc, $mc) = (0, 0);
for my $line (‘dir/prot/width=(file=60) [...]‘)
{
  my ($file, $w) = $line =  m/(.+)\s+.+,(.*)\)/;
  next unless $file;
  $fc++;
  print "$file\n" and $mc++ if ($w =  m/[WD]/);
}

print "$fc files processed, $mc are world
    writable/deletable!\n";
```

Migrating a MySQL database to RDB

On one occasion I had to migrate quite a lot of data from a MySQL database running on a LINUX system to an RDB database running on an OpenVMS system.

The first idea was to write a database dump from the MySQL system, reformat this using Perl into something which could be understood by RDB and feed the resulting data into the RDB system.

This turned out to be too cumbersome, so another approach was taken: Write a short Perl program connecting to both databases and copying data on the fly from one system to the other.

Migrating a MySQL database to RDB (initialization)

```
use strict;
use warnings;
no warnings qw /uninitialized/;
use Net::MySQL;
use DBI;
use DBD::RDB;

my $rdb = DBI -> connect (
  'dbi:RDB: ATTACH ALIAS RECIPES FILENAME
  DISK$RDB_DATA:[000000]RECIPES', undef, undef,
   RaiseError => 1, PrintError => 1, AutoCommit => 0, ChopBlanks => 1 );

my @tables = qw/art eigenschaften einheiten glutenfrei kategorien
    personen region rezept_kategorien rezept_zutaten recipes zustand
    zutaten/;

my $mysql = Net::MySQL -> new (hostname => 'klapauzius.pi-research.de',
    database => 'recipes', user => 'rikka',
    password => ':-)');
```

Migrating a MySQL database to RDB (copying data)

```
for my $table (@tables)
{
  $rdb -> do ("delete from recipes.$table");
  $mysql -> query ("select * from $table");
  my $record_set = $mysql -> create_record_iterator;
  my @fields = $record_set -> get_field_names;
  my $statement = "insert into recipes.$table (" .
        join (',', @fields) .  ')  values (' .
        join (',', map {'?'} 0..$#fields) .  ')';
  my $rdb_sth = $rdb -> prepare($statement);
  my $counter = 0;
  while (my $record = $record_set -> each)
  {
    $rdb_sth -> execute(@$record);
    $rdb -> commit unless $counter % 50;
  }
  $rdb -> commit if $counter % 50;
  print "\nInserted $counter lines into recipes.$table\n";
}
$rdb -> disconnect;
```

Larger programs

Many problems which occur on a regular basis can be solved using Perl, too. Examples for such problems are:

- Generating simple web server statistics on a daily basis.
- Fetching mail from a POP server in regular time intervals and distributing these mails to the OpenVMS mail system.
- Sending outgoing mails to an SMTP server requiring authentication which is not currently supported by OpenVMS.
- Caching results from database queries to speed up execution time of programs requesting data from the database.

These examples will be briefly described in the following.

Simple web server statistics

After observing that the webserver running on my OpenVMS system is rather busy serving requests I wanted to have a simple web server statistics to know which files are requested how often from remote users. All in all I wanted to generate a log file like this every night in a small batch job:

```
2734:  my_machines/dornier/do80/
 288:  my_machines/bbc/tisch_analogrechner/
 117:  publications/anhyb.pdf
  97:  publications/handson.pdf
  88:  my_machines/eai/understanding/underst_analog_hybrid_comp.m4a
  83:  analog_computing/vehicle_simulation/weak_damping.avi
  72:  my_machines/dornier/do80/do80_bedienungshandbuch.pdf
```

Simple web server statistics

```
use strict;
use warnings;
die "File name and account name expected!\n" unless @ARGV == 2;

my ($log_file, $account) = @ARGV;
open my $fh, '<', $log_file or
  die "Unable to open log file $log_file, $!\n";
my %matches;
while (my $line = <$fh>)
{
  my ($ip, $key) = $line =~ m/^(\d+\.\d+).*$account\/(.+?)\s/;
  next if !$ip or $ip eq '192.168';
  $key =~ s/"//g;
  $matches$key++ if $key;
}
close $fh;

printf "%5d:  %s\n", $matches{$_}, $_
  for (sort $matches$b <=> $matches$a keys(%matches));
```

Fetching mail from a POP-server

Sometimes it is desirable to fetch mails from a typical POP3-server and make these available in the OpenVMS mail system so the system users can access their mails using `MAIL` or a suitable web interface like `yahmail` or `soymail` etc.

To make this possible, a Perl written batch job is required which polls in regular intervals a variety of POP3-servers and mailboxes, fetches mails and distributes these mails to the various users of the OpenVMS system.

Fetching mail from a POP-server

Using the following modules, the overall Perl code for implementing this batch job consists of only 140 lines:

Net::POP3;	Client interface to the POP3-protocol.
IO::File;	File creation and access methods.
POSIX qw(tmpnam);	Used for creating temporary file names.
VMS::Mail;	Interface to the OpenVMS mail system.

When it is possible to receive mails, it would be nice to be able to send mails, too, as the following slides show.

SMTP-proxy

Almost every current mail provider requires its clients to authenticate prior to sending mail via their SMTP server(s).

Unfortunately authentication is not supported by the TCPIP package for OpenVMS. Since I had the requirement to send outgoing mail directly, i.e. without an intermediate system, it was decided to implement a small SMTP-proxy running on the OpenVMS system.

This proxy connects on the local machine to port 25 and listens for outgoing mail. Another connection is made to port 25 of the provider.

SMTP-proxy

Every outgoing mail is parsed and enriched with the necessary authentication information before being sent to the provider which solved the problem quite easily.

The SMTP proxy makes use of the following modules which results in an overall code size of only 68 lines of Perl:

`Net::ProxyMod;`	This module allows easy packet modification.
`MIME::Base64;`	MIME-encoding and -decoding
`Tie::RefHash;`	Allows using references as hash keys.

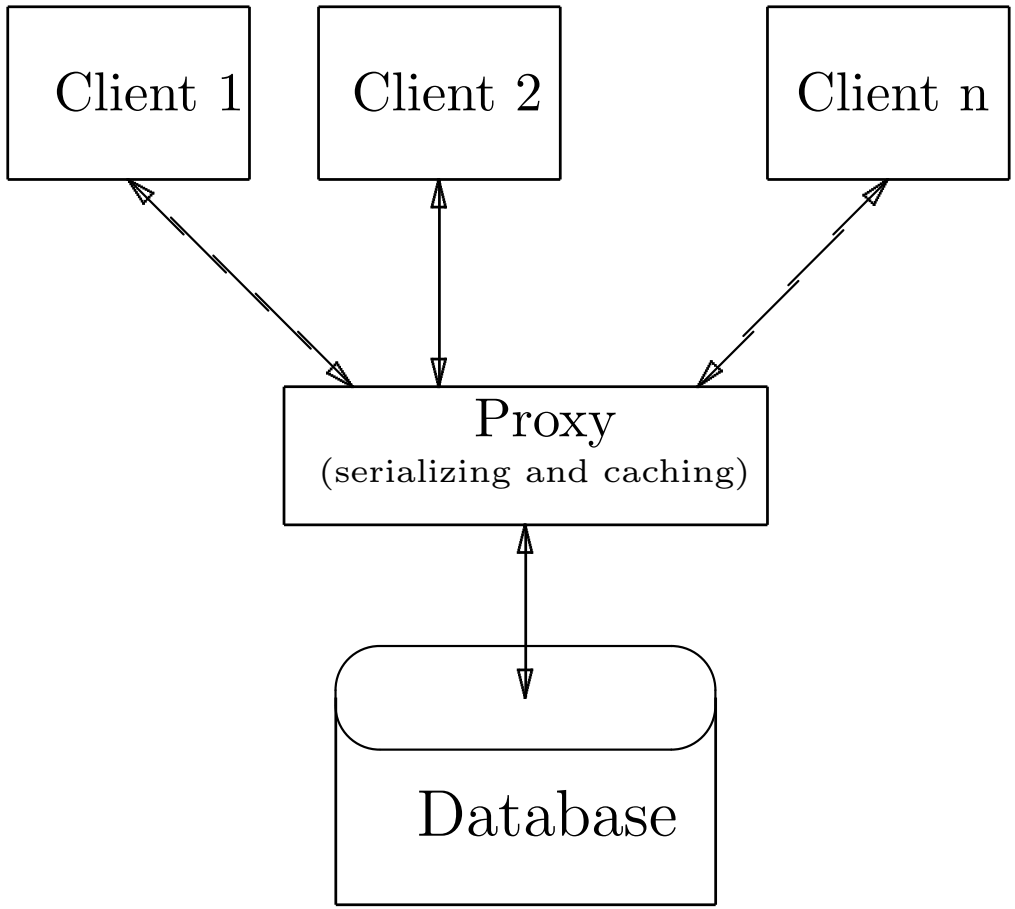
A database proxy

Sometimes it is desirable to perform database accesses not directly but via a proxy which might either contain some business logic and/or cache as much data as possible to reduce database load at the cost of some additional memory consumption.

In 2005 the problem arose that accesses to an RDB database running on a VAX were too slow and did not match expectations.

Since all of these accesses were reads containing several joins and the like, the idea of implementing a proxy server in Perl to cache the results yielded by such accesses and deliver the cached results for all following accesses instead of querying the database came up.

A database proxy



A database proxy

The speedup effect which can be obtained using this Perl written data base proxy is dramatic:

A typical request which takes 3.2 seconds when issued directly to the database is satisfied in 3.52 seconds using the proxy with empty cache (i.e. directly after startup).

The very same request issued to the proxy which already has the required data in its cache takes only 0.34 seconds and is thus roughly ten times faster than the direct database access.

Of course tuning the proxy with respect to its cache size and cache lifetime depends on the type of workload.

Conclusion

- Perl turns out to be an invaluable tool for everyday usage as well as for large and complex production programs.
- Especially in an OpenVMS environment which often has special needs when it comes to system connectivity and the like, Perl can be employed with much benefit.
- Perl does not consume too many resources and is really fast for an interpretive language, so it can be even used on smaller VAX systems.
- It is important to realize that Perl is not a *scripting language* but rather a very mighty programming language. Thus Perl should be taken seriously.

Contact

The author can be reached at

ulmann@vaxman.de