Perl on MVS
and
the Art of Parsing and Generating SWIFT messages

Bernd Ulmann*

July 18, 2006

Abstract

The following paper will not present new Perl stunts, etc. – it is merely a report covering some experiences gained in introducing and using Perl on an IBM mainframe running z/OS. It was decided to use Perl on this platform to implement interfaces for sending, receiving and processing SWIFT messages used in interbank communications.

This paper gives an account of the problems faced during this development (there were quite a few) and the solutions and work arounds used to resolve these.

1 Introduction

In 2005 it was decided that it would be crucial for the future business of the bank I am working for to be able to receive, parse, process and generate SWIFT messages[1] of several different types (so called *Message Types*, *MT* for short).

The messages in question for our application are used for interbank communications between capital investment banks, brokers and depositary banks. Figure 1 shows the message flow between these parties:

1. When a capital invesment bank wants to sell or buy some stock for a deposit account it will first involve one of its brokers by sending him appropriate instructions via FIX, by telephone, fax, etc.
2. The broker will then make the trade requested and will inform the capital investment bank about this action by generating and sending an MT515[2] message.
3. This message will be parsed by the capital investment bank and will form the basis of an MT541[3] or an MT543[4] message (depending on buy or sell) which will be sent in turn to the depositary bank. This bank will parse this message and create a trade in its systems reflecting the deal made by the broker.

*ulmann@vaxman.de

[1] In 1973 several large financial institutions (239 to be exact) from fifteen countries founded a society to facilitate the interbank transfer of business data, which became known as *SWIFT* (short for *Society for Worldwide Interbank Financial Telecommunication*). The heart of SWIFT is the so called SWIFT network, a dedicated network which transmits millions of message per day. The messages transferred have to comply with strict format rules and are denoted by a number each representing their respective type and usage.

[2] *Client Confirmation of Purchase or Sale*

[3] *Receive Against Payment*

[4] *Deliver Against Payment*

4. After making the trade the depositary bank will send an MT545[5] or MT547[6] message back to the capital investment bank thus informing them that the trade has been successfully made.
5. The capital investment bank can order a *Statement of Holding* from their depositary bank showing all of their deposit accounts – this statement is sent in form of an MT535 message on a daily, weekly, monthly or quarterly basis.

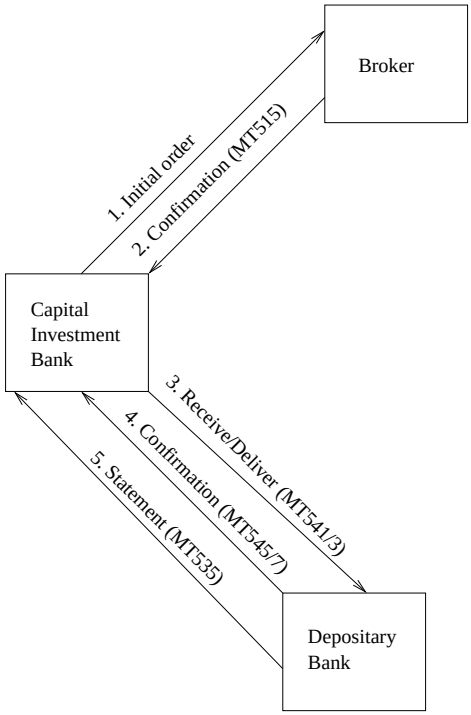

Figure 1: SWIFT messages in the depositary bank environment

We were especially interested in the processing of the MT541, MT543, MT545, MT547 and MT535 SWIFT messages. This led to the creation of a project with the aim of implmenting the necessary software to receive, send, create and parse such messages.

1.1 Format of SWIFT messages

Having a look at the format of a typical MT541 message it becomes quite clear that parsing a SWIFT message is not much fun at all if you are forced to do it the traditional mainframe way, using COBOL or even Assembler. Figure 2 shows a typical MT541 message (made anonymous).

The first obvious thing about a message like this is its bracket structure:

```
{1:...}{2:...}{3:...}{4:...}{5:...}
```

The first bracket contains the recipients address, together with some information about the urgency of the message, etc. The second block carries information about the sender while the third block is an optional user header. The main message is

[5] *Receive Against Payment Confirmation*
[6] *Deliver Against Payment Confirmation*

```
{1:F01NO314BICB5507798107552}
{2:O5411443060705SENDERBICXXX57299347020607051443N}
{3:
{108:2752159.541.01}}
{4:
:16R:GENL
:20C::SEME//3141592653
:23G:NEWM
:16S:GENL
:16R:TRADDET
:98A::TRAD//20060704
:98A::SETT//20060706
:90B::DEAL//ACTU/EUR27,180000
:35B:ISIN XU9786546231
PI Research Inc.
:16S:TRADDET
:16R:FIAC
:36B::SETT//UNIT/96,
:97A::SAFE//1729B
:16S:FIAC
:16R:SETDET
:22F::SETR//TRAD
:22F::DBNM//INTE
:16R:SETPRTY
:95Q::DEAG//Large Bank Inc.
CBF 1234
D-Frankfurt am Main
:16S:SETPRTY
:16R:SETPRTY
:95P::SELL//LBNKDEFF
:16S:SETPRTY
:16R:AMT
:19A::SETT//EUR75876,16
:16S:AMT
:16S:SETDET
-}
{5:
{MAC:E9632025}
{CHK:9ABBEB5739EE}}
```

Figure 2: Example of a rather short MT541 SWIFT message

embedded in the fourth block – this is the really interesting part, since it contains the trade information which will be used to actually create or cancel a trade.

Having a look at the fourth block of such a message it becomes clear that this part is block structured, too. A block is opened with `:16R:block_name` and closed by `:16S:block_name`. Blocks can be nested (and usually are), but the order of this block sequence is of prime importance.

Parsing something like this with any language but Perl did not seem like a good idea, so it was decided to use Perl for this task. Since our connection to the SWIFT network is implemented using a software called Merva which runs on our z-series mainframe under z/OS, it was decided to use Perl in this quite unusual environment.

2 Perl on MVS

Fortunately there is a Perl port for z/OS but this interpreter can not run in the native z/OS environment. Instead it relies on the existence of the so called USS environment, the *UNIX System Services*. This environment is run as an address space under the native z/OS system and provides a decent UNIX environment suitable to feeling a bit at home on such a strange machine.

2.1 Compiling and linking the Perl interpreter

Since IBM only offered a very old Perl distribution at the time we decided to use Perl on our mainframe, we had to compile our own Perl 5.8.6 on the machine which turned out to be not that easy at all[7].

The first thing you will be realize when working in the USS environment is that some UNIXes are different. This is *very* different – from every other UNIX I have worked with so far. First of all, there are nearly no tools installed at all. To begin working you will need at least `gzip`, `make-1.76` (the IBM supplied `make` was not able to compile the Perl interpreter and generated lots of quite strange error messages) and maybe `groff`. If you are working in a restricted environment (as is usual in banks) I suggest installing these tools locally and modifying your `PATH` variable accordingly.

To unpack the Perl distribution you will have to transfer it to a local directory and then run `gzip -d file_name`. Since the resulting `.tar`-file will contain ASCII data while the mainframe is an EBCDIC machine, you can not simply use `tar` for unpacking it. Instead use `pax -o to=IBM-1047,from=ISO8859-1 -r < file_name`.

Before attempting to build Perl it should be taken into account that even a modern z-series system is fairly slow compared with other state of the art (UNIX) systems. You will experience long run times (up to several hours) and you will have your administrators set the following limits for your USS-account[8]:

1. `SETOMVS MAXASSIZE=671088640`
2. `SETOMVS MAXCPUTIME=10000000`

After these preliminaries, `sh ./Configure` can be run which will generate a `WHOA`-message – keep the `#define`-statement mentioned there. Do not even think about compiling with `-O` – optimized code does not seem to work on a z-series system. `make test` will hang in the UDP test.

2.2 Building `DBI` and `DBD`

Compiling `DBI` and `DBD` especially is a bit more complicated than building the Perl interpreter itself.

2.2.1 `DBI`

Just extract the sources with `gzip` and `pax`, then perform the following steps:

1. `perl Makefile.PL`
2. `make -f Makfefile`
3. `make -f Makefile perl` – this step takes a very long time to complete!
4. `make -f Makefile.aperl MAP_TARGET=perl`
5. `make -f Makefile install`

2.2.2 `DBD`

`DBD` is more of a challenge. First of all you will need the required DB2 header files to compile the C sources. Since these do not normally reside in the USS file system, you will have to copy them from the MVS system to the USS environment. These

[7] In the meantime IBM offers a current distribution, too, so things may be easier now, but we have to stay with our version for some time at least.

[8] I am sure that lower limits would work, too, but we did not have time to investigate this point on our system

files will reside in a partitioned dataset like DB2S710.SDSNC.H. The natural way to copy these files would be using the command oget – using ftp is much more simple.

Be sure to add the path of the directory where these header files reside to your PATH variable. In addition to that you will need to create an environment variable named DB_HOME containing the very same path information.

The build process is quite difficult (in my oppinion):

1. cd Constants
2. perl Makefile.PL
3. make -f Makefile perl
4. make -f Makefile.aperl inst_perl MAP_TARGET=perl
5. make -f Makefile install
6. cd ..
7. perl Makefile.PL – after this completes, you will have to change the entry CC in line 32 of Makefile from c89 to c89 -W c,dll. Then run
8. make -f Makefile perl – this step will result in a return code 8 from the LINKEDIT step, which is perfectly normal and nothing to worry about. This will be resolved in the following step:
9. Using a suitable editor perform the following changes in Makefile.aperl:
   (a) Change MAP_LINKCMD=$CC to MAP_LINKCMD=c89 -Wl,p,dll,AMODE=31.
   (b) Extend the line reading MAP_PRELIBS=-lm -lc" to MAP_PRELIBS=-lm -lc "//'DB2SYS.SDSNMACS(DSNAOCLI)'".

   Then you will have to inform c89 about the fact that DB2SYS.SDSNMACS has no extension .EXP as it would normally be expected by issuing export _C89_XSUFFIX_HOST="SDSNMACS".
10. The rest is business as usual: make -f Makefile perl, make -f Makefile.aperl inst_perl MAP_TARGET=perl and, finally, make -f Makefile install.

2.3 Accessing DB2

All in all accessing DB2 from within a Perl program running in the USS environment is very straightforward.

```
use DBI;
use DBD::DB2::Constants;
use DBD::DB2;

my ($username, $password) = ('DB2USR', 'SOMEPASSWORD');
my $dbh = DBI -> connect ("dbi:DB2:ENV", $username, $password) or
  die "Could not connect to database! $!\n" if !defined ($dbh);

my $sth = $dbh -> prepare (...);
$sth -> execute (...) or die "PANIC!\n";
$sth -> finish;

$dbh -> disconnect;
```

Assuming that the code fragment above is run in the context of a user BATUSR and the connection to DB2 is made as the user DB2USR the following settings are required to successfully run the program and access the database:

1. The user BATUSR needs execute right granted for the plan DSNACLI,
2. the user DB2USR needs the necessary select-, insert-, update- or delete-rights on the tables, views, etc. which are involved with the application.

3. Finally `.profile` of the user `BATUSR` must contain a line like this (depending on your installation):

```
export STEPLIB=xxxxxxxx.SDSNEXIT:xxxxxxxx.SDSNLOAD
```

2.3.1 Writing long character fields

Using `DBI` and `DBD` in the USS environment with DB2 running on z/OS you can not insert values exceeding 255 bytes into table rows at once. This turned out to be a severe problem for our application since we have to insert complete SWIFT messages into some tables using `VARCHAR(32000)` columns.

Let `$msg` contain a string exceeding 255 bytes in length. To successfully insert this into a row of a DB2 table from a Perl program running in the USS environment you have to split this string into substrings of 253 bytes at most which are then concatenated with `||`.

So instead of performing

```
INSERT INTO TABLE ('MESSAGE')
VALUES ('A VERY, VERY, VERY LONG MESSAGE')
```

you have to do something like this:

```
INSERT INTO TABLE ('MESSAGE')
VALUES ('A VERY, '||'VERY, VERY '||'LONG MESSAGE')
```

This rose the question of how to split long strings into parts of constant length (apart from the last snippet, of course). Running

```
use strict;
use warnings;
use Benchmark qw(cmpthese);

my $x = 'a' x 5000000;
my $chunk_size = 253;

my %bench = (
  _unpack => sub
    {
      my @parts = unpack("A$chunk_size" x
                         (int(length($_[0]) / $chunk_size) + 1), $_[0]);
      return @parts;
    },

  _substr => sub
    {
      my($str) = @_;
      push my @parts, substr($str, 0, $chunk_size, '') while $str;
      return @parts;
    },

  _map => sub
    {
      my @parts = grep { $_ ne '' } split /(.{$chunk_size})/, $_[0];
      return @parts;
    },
);
```

```
cmpthese($ARGV[0] || 1000000, \%bench);
```

on our mainframe yields

```
            Rate _unpack    _map _substr
_unpack  90009/s      --    -54%    -67%
_map    194175/s    116%      --    -30%
_substr 275482/s    206%     42%      --
```

showing that the simple `substr`-solution performs fastest. With this in mind we wrote a simple function which we use prior to any insert operations to split long strings into 253 byte chunks concatenated by `||`.

2.4 Running Perl from JCL

A normal mainframe environment will require all programs to be controlled from the z/OS environment, so a suitable JCL wrapper for running Perl programs in the USS environment is needed. Apart from just running a program is will be necessary to make `stdout` and `stderr` from the USS available to the z/OS side of the system.

This turns out to be a bit tricky – especially if you are not too fluent in JCL. The following JCL fragment might serve as a template for own experiments:

```
//PERLTEST JOB 'TEST','USER',MSGCLASS=M,MSGLEVEL=(1,1),
//         USER=&SYSUID,NOTIFY=&SYSUID,CLASS=T
//*****************************************************************
//*
// SET SYS='development'
// SET DIR='test'
// SET PRG='db2_test.pl'
//*
//*****************************************************************
//*
//RUNPERL  EXEC PGM=BPXBATCH,
//   PARM='sh perl /usr/local/&SYS/&DIR/&PRG..pl &SYS..ini'
//*
//* Set environment variables
//STDOUT   DD   PATH='/tmp/&PRG..out',
//         PATHOPTS=(OCREAT,OTRUNC,OWRONLY),PATHMODE=SIRWXU
//STDERR   DD   PATH='/tmp/&PRG..err',
//         PATHOPTS=(OCREAT,OTRUNC,OWRONLY),PATHMODE=SIRWXU
//*
//* Since BPXBATCH can only redirect stdout and stderr into
//* HFS-files these will now be copied back to the MVS environment
//CPOUT    EXEC PGM=IKJEFT01,DYNAMNBR=300,COND=EVEN
//SYSTSPRT DD  SYSOUT=*
//HFSOUT   DD  PATH='/tmp/&PRG..out'
//HFSERR   DD  PATH='/tmp/&PRG..err'
//STDOUTL  DD  SYSOUT=*,DCB=(RECFM=VB,LRECL=133,BLKSIZE=137)
//STDERRL  DD  SYSOUT=*,DCB=(RECFM=VB,LRECL=133,BLKSIZE=137)
//STDOUTLF DD  DSN=&&STDOUTLF,
//             DISP=(NEW,PASS),
//             SPACE=(CYL,(20,10),RLSE),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=0)
//STDERRLF DD  DSN=&&STDERRLF,
//             DISP=(NEW,PASS),
//             SPACE=(CYL,(20,10),RLSE),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=0)
```

```
//SYSPRINT DD  SYSOUT=*
//SYSTSIN  DD  *
OCOPY INDD(HFSOUT) OUTDD(STDOUTL)
OCOPY INDD(HFSOUT) OUTDD(STDOUTLF)
OCOPY INDD(HFSERR) OUTDD(STDERRL)
OCOPY INDD(HFSERR) OUTDD(STDERRLF)
//*
//* If something went wrong, perform some recovery actions.
// IF RC NE 0 THEN
... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ...
// ENDIF
```

3 Interfacing to the SWIFT network and handling SWIFT messages

3.1 Connecting to the SWIFT network

Making the connection to the SWIFT network for incoming and outgoing messages is by far the most difficult part of a project like this. Since we use a rather old interface software called Merva, which has been in use in our company for years and years to actually connect to the SWIFT network, we were in need of a generic interface between Merva and Perl.

We decided to write a DB2 UDF using Assembler (this has historical reasons which you do not want to know about) and C. Then we set up a table which mainly contained only a single row called `MESSAGE` of type `VARCHAR(32000)`. This table, called `OUTGOING`, has an associated trigger which fires every time a new row is inserted. This trigger in turn starts the UDF which actually transfers the data read from the rows inserted to our Merva system.

The other way around was done quite similar – this time we have something like a trigger in Merva which will write incoming messages in raw format (i.e. as a long string with line delimiters) into another database table called `INCOMING`. Figure 3 shows these data paths in our system.

Using these two techniques accessing the SWIFT network from Perl is as simple as inserting a row into the table `OUTGOING]` or reading a row from `INCOMING`. Having implemented this the next thing to do is to parse and generate SWIFT messages.

3.2 Parsing SWIFT messages

According to John Davies[9] and others, parsing SWIFT messages is a difficult task:

> I'm not a betting man but I would put serious money on the fact that even the brightest of programmers could not write a reliable SWIFT parser for any given message type in under a week. Take an "average" programmer though and you're looking at several weeks to get it right.

When it comes to the processing of a SWIFT message like the one shown in figure 2 there are at least two useful possible solutions:

1. Parse the message as a whole and translate it to a nested data structure.
2. Extract only the data needed and discard everything else.

The latter approach is by far not as generic as the first one but way simpler (using a hash of hashes to store such a message would not suffice since the order of the sublists is crucial for SWIFT), so it was decided to take this approach.

[9] `http://www.c24.biz/download/c24_white_paper-parsing_a_swift_message.pdf`

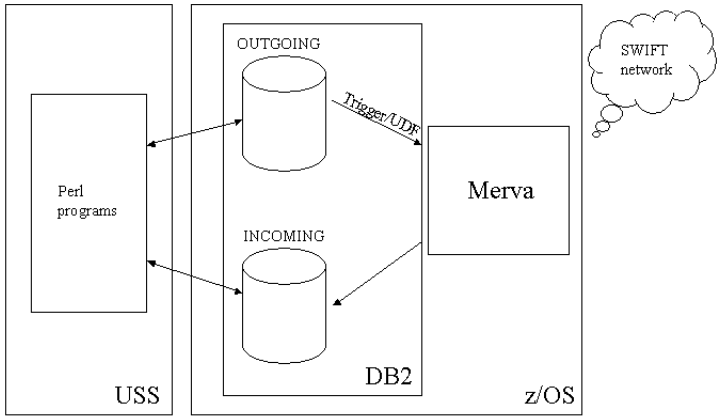

Figure 3: Connecting to the SWIFT network

Since the messages itself are quite fixed in their format and only a couple of fields (about a dozen) of the message above are really needed for further processing, these are extracted into a hash using appropriate field names (`$msg` will contain the raw message as a string). The following example will extract the trade quantity and the quantity type from a MT541/543 message:

```
my %data;

($data{QUANTITY_TYPE}, $data{QUANTITY}) =
  $msg =~ m!:16R:FIAC\n.*?36B::SETT//([A-Z]{4})/([^\n]+)!s;
```

Note that it is important to use frugal matching in most of these regular expressions since many sequences may occur more than once in a message which leads to funny results at best if greedy matching is employed.

Using this technique we were able to write a reliable parser for MT541/543 messages with two persons working for one day – thanks to Perl's powerful pattern matching. The fields extracted into our flat hash are subsequently used to create trades in out front office system which involves quite a lot of code to check for the existence of instruments to trade on, etc.

3.3 Generating SWIFT messages

Since the structure of SWIFT messages is rather fixed – in fact the whole sequence of fields is fixed, only some fields and blocks may be omitted – we decided to use a very simple method of creating SWIFT messages from within Perl: We just use several `sprintf` statements with some logic buried in `?`-operators and surrounding `if`-statements.

The main pitfall is the way SWIFT expects signed numerical data with an associated currency like -1000000.00 EUR. Due to some arkane reasons a valid SWIFT message would contain a string of `NEUR1000000,00` representing the value above. Positive signs are omitted while negative signs are represented as `N` preceding the currency which is then followed by the absolute value.

As soon as a SWIFT message has been created it is sent to the SWIFT network by inserting it into the `OUTGOING` table triggering the trigger which starts the UDF which finally transfers the message to Merva which makes the actual connection to the SWIFT network.

4 Conclusion

The decision to make Perl available on our mainframe was crucial in getting the already mentioned SWIFT project completed in time. Yet it was not that easy to create a suitable working environment on the mainframe, but the results were very well worth the efforts. And it turned out that other developers welcomed the availability of Perl in this environment. This makes even a rather arkane system like a mainframe running z/OS interesting for younger programmers, too.

As expected it turned out that productivity boosted after Perl was available. Compared with traditional mainframe languages like Assembler, COBOL and even REXX, Perl is far ahead when it comes to ease of use and semantical power. We even used our Perl installation to replace some simple interfaces written in traditional languages with Perl code.

Unfortunately there are still some unresolved problems as today:

1. We could not get `Net::SMTP` to run – there seems to be a problem with the EBCDIC support on MVS.
2. A more severe problem is that the VSAM-access method of the `OS390::Stdio` packet does not work either.
3. An early attempt to install XML-support failed due to some problems with the EBCDIC support.