# Perl on z/OS and the Art of Parsing and Generating SWIFT Messages

Bernd Ulmann
ulmann@vaxman.de

YAPC::Europe 2006
30th August – 1st September 2006
Birmingham

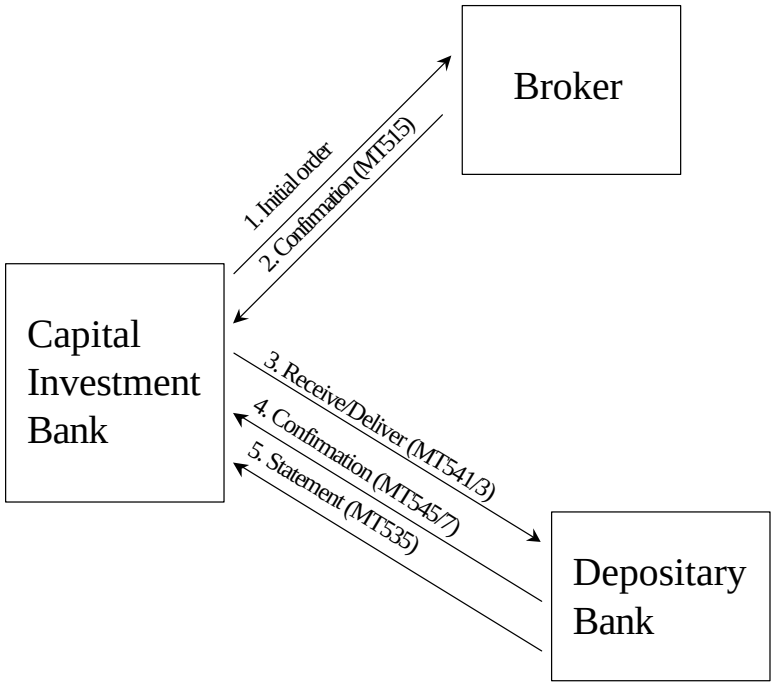
# What is SWIFT?

- Founded in 1973 by 239 financial institutions.
- Short for *Society for Worldwide Interbank Financial Telecommunication*.
- The heart of SWIFT is a special purpose network to transmit data for interbank communication (guaranteed delivery).
- Transmits several millions of messages per day.
- Messages have to comply with strict format rules.
- Literally hundreds of different message types (*MT* for short).

# SWIFT for depositary banks

- In 2005 my employer – a financial institution – decided that being able to send, receive, generate and parse SWIFT message automatically would be crucial for the future depositary bank business.
- This got me involved with the overall SWIFT theme since I became project manager for this endeavor.
- Since our connection to the SWIFT network and many other things are done on a large z-series mainframe running z/OS, the idea of using Perl in this rather strange environment came up.
- The following picture shows the flow of information in the depositary bank business:

# SWIFT for depositary banks



# How does a typical SWIFT message look like?

- Horrible!
- It looks a bit like XML, but only a bit...
- The next slide shows a real SWIFT message used to create a trade (made anonymous):

# Format of SWIFT messages

```
{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
:16R:SETPRTY
:95P::SELL//LBNKDEFF
:16S:SETPRTY
:16R:AMT
:19A::SETT//EUR75876,16
:16S:AMT
:16S:SETDET
-}
{5:
{MAC:E9632025}
```

# Properties of a SWIFT message

- Block structured with brackets: $\{\ldots\}\ldots\{\ldots\}$.
- Block 4 contains the actual data.
- This data block is itself block structured:

```
:16R:name
...
...
:16S:name
```

- Sequence of blocks and items in blocks is important!
- SWIFT has an interesting way to denote negative values.
- Parsing such a message is not too simple.

# Background information

Since parsing a SWIFT using COBOL or Assembler (this is not a joke!) is no fun at all, it was decided to use Perl on the mainframe to get the project finished in time.

- Perl cannot run on the native z/OS environment.
- Perl for z/OS runs in the UNIX-System-Services environment (*USS* for short).
- Now there is a prebuilt Perl 5.8.7 available from IBM – this was not the case when we started, so we built everything ourselves.

# Prerequesites

- Even a decent z-series system is quite slow compared to current UNIX systems when it comes to building Perl – expect compile times of several hours!
- The build process will require higher memory and CPU-time limits than normally available in a mainframe shop:

```
SETOMVS MAXASSIZE=671088640
SETOMVS MAXCPUTIME=10000000
```

- You will have to install `gzip` and `make-1.76` locally. The IBM supplied `make` utility breaks the Perl build process!
- Due to the EBCDIC nature of the z-series you cannot use `tar`, instead you have to use `pax` like this:

```
pax -o to=IBM-1047,from=ISO8859-1 -r < file_name
```

# Building Perl

Building Perl is quite straight forward as long as you take the following two items into account:

- Do not even think about compiling with `-O` – the resulting code will just not run!
- Keep the `#define`-statement which is mentioned by the `WHOA`-message thrown by running `./Configure`.

# Building DBI

Building DBI is straight forward – just follow the documentation:

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

# Building DBD

Building DBD is more of a challenge:

- You will need the required DB2 header files which normally reside in a partitioned dataset on the z/OS side of the system (e.g. `DB2S710.SDSNC.H`).
- You can copy these files with `oget` or just using `ftp` for those who are not too familiar with the arkane z/OS side.
- Place these files in a subdirectory in the USS environment and extend your `PATH` variable to contain this directory as an entry. Furthermore you will need an environment variable `DB2_HOME` pointing to this directory, too.

# Building DBD

The actual build process is quite difficult:

- `cd Constants`
- `perl Makefile.PL`
- `make -f Makefile perl`
- `make -f Makefile.aperl inst_perl MAP_TARGET=perl`
- `make -f Makefile install`
- `cd ..`

# Building DBD

- `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
- `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:

# Building DBD

- Using a suitable editor perform the following changes in `Makefile.aperl`:
  - Change `MAP_LINKCMD=$CC` to `MAP_LINKCMD=c89 -Wl,p,dll,AMODE=31`.
  - 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"`.
- 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`.

# Accessing DB2

Accessing a DB2 system running on z/OS from a Perl program running on USS is straight forward.
Assuming that the program runs in the context of a user called `BATUSR` and uses a database user `DB2USR` to connect, the following rights must be granted:

- `BATUSR` needs execute right for the plan `DSNACLI`.
- `DB2USR` needs the necessary rights for the database objects he wants to access.

# Inserting large character fields

Inserting character fields exceeding 255 bytes in length is a bit of a problem – it just does not work out of the box.
The workaround we resorted to use is to split large strings in 253 byte chunks and concatenate these explicitly using ||:

- Instead of issuing

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

- we do

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

# Inserting large character fields

To facilitate the splitting into chunks we use a function like this:

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

# Running Perl from JCL

- JCL, the *Job Control Language* is the heart of every job running on a z/OS system.
- Even if a Perl program is running on USS it is quite likely that it will be controlled from within a JCL environment.
- Running such a program involves more than just starting a task:
  - Execute the Perl program in the USS environment with output redirection using `BPXBATCH`.
  - Define characteristics of datasets to hold `stdout` and `stderr`, so that these will be available on the z/OS side.
  - Copy the redirected output to the datasets defined before.
  - Interpret the return code and take appropriate actions.

# A JCL example – part 1

```
//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
//*
```

# A JCL example – part 2

```
//* 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
```

# Sending messages to the SWIFT network

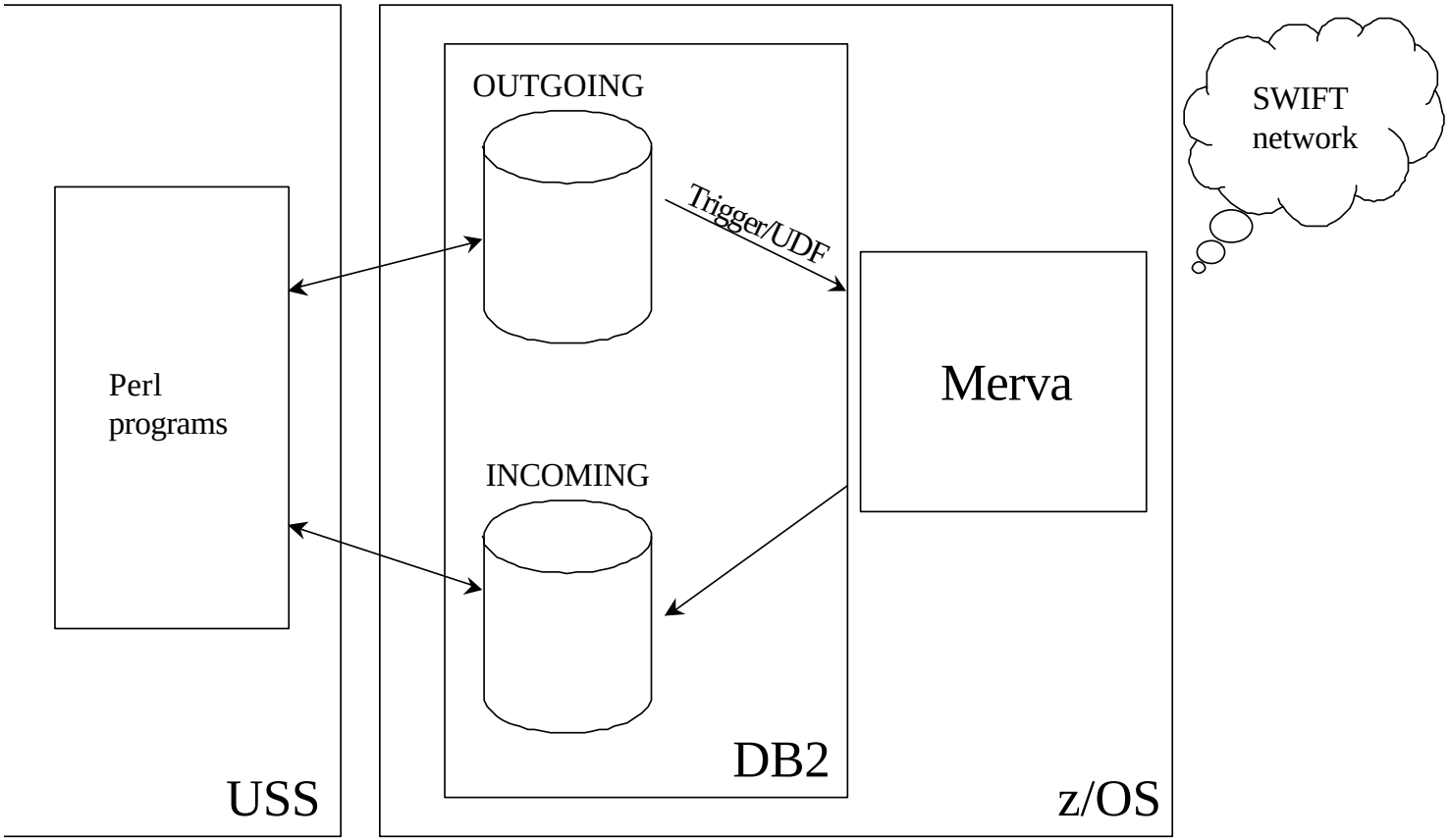
- We use a rather old software called "Merva" as the interface to the SWIFT network. Merva runs in the CICS environment of the mainframe and has no simple interfaces for connecting inhouse developed applications easily.
- Since there was an Assembler code snippet which could send a SWIFT message to the Merva system already in production, we decided to make use of this, too.
- We wrote a DB2 UDF using C which calls this Assembler program with a SWIFT message as an argument.
- This UDF will be called by an insert trigger defined on a DB2 table in the DB2 system. Inserting a row into this particular table will fire the trigger, start the UDF and transmit the SWIFT message to Merva and thus to the SWIFT network.

# Receiving messages from the SWIFT network

The other way around works quite the same:

- Reusing another small Assembler program we created something like a trigger in Merva which is fired every time a SWIFT message of a particular message type is received.
- This trigger will then connect to the DB2 system and insert a row into a table for every message received this way.
- The following picture shows the overall architecture of this inhouse developed interface to the SWIFT network.

# Using DB2 trigger and UDFs as an interface to SWIFT



# Generating SWIFT messages

Since SWIFT messages have a very rigid structure where the position of every single field is fixed, we decided to create such messages by using simple `print` statements and the like:

```
$subsequence_d3 = "
:16R:AMT
:19A::DEAL//$data->TRADCIRR$nominal
:16S:AMT
:16R:AMT
:19A::EXEC//$data->TRADCURR$data->DMG1BT_4
:16S:AMT" if $amounts;
```

# Just how hard can it be?

According to John Davies[1] 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.*

. . . obviously he is a Java programmer!

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

# . . . it depends. . .

- Writing a generic parser for SWIFT messages would not be a simple task, but. . .
- We have only two message types which have to be parsed – the rigid message structure makes parsing easy when you know what you are looking for.
- Using Perl it took two persons (one programmer and one trainee) a mere day to write a simple parser suitable for use in a production environment!
- Remember the time estimated time frame above?

# Extracting data

- We decided to extract only the necessary fields into a flat hash structure.
- Due to the repetitive nature of substructures in SWIFT messages it is important to use frugal matching!
- The following example is used to extract the trade quantity and the quantity type from a MT541 message ($msg contains the raw message data, %data is the hash mentioned above):

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

# Conclusion

- Running Perl on a z/OS machine is not too easy in the beginning (this might have changed in the meantime), but is worth the effort.
- Productivity boosts:
  - We even began to replace some programs written in COBOL and the like with short and maintainable Perl programs.
  - The availability of Perl makes the mainframe more interesting for young programmers, too, who are not too familiar with z/OS and do not really like to think in punch card formats.
- Some problems are still unresolved:
  - We could not get `Net::SMTP` to run due to problems with EBCDIC support.
  - An attempt to install XML-support failed, too.
  - The most severe problem for us is that the VSAM-access method of the `OS::Stdio` packet does not work.