**A SummeRx White paper.**

REXX is a straightforward old-fashioned language that aims to write programs that manipulate z/OS system resources relatively quickly.

Some of the big limitations of REXX, such as the inability to pass stems as arguments to external routine or to expose them as shell variables, have led to a situation where the power of REXX and its ease of writing is mostly used today in exits or conversion processes that do not require many lines of code, although there are many older applications that need to be supported. Indeed, for the new generation of programmers who have learnt OOP in universities, maintaining large systems written in REXX can be a challenge.

Although there are many system functions available in REXX through z/OS commands, the inconvenience of programming with primitives and the lack of modularity make REXX a less popular tool on the market.

SummeRx, which was developed in Sailpoint as part of a large code change management project, initiated a new approach to writing in REXX and then continued as a convenient platform of REXX that supports dynamic modularity, HEAP, templates, powerful messaging and other functions.

Can you imagine that the REXX code of a TCPIP server that supports multiple clients will be about 20 lines? Did you expect to be able to write REXX programs that implement provider-consumer functionality? And many other features that you could not even imagine.

So, let me introduce the SummeRx framework. It is fully written in REXX, has open source code and initiated for future REXX contributions by programmers and is expected to expand with other environments such as DB2, Linux, Dockers, etc...

The heart of SummeRx is an injector. Injector is dynamically called from SummeRx REXX program. SummeRx REXX program is a REXX program written according to a certain template, which allows the injector to start at the very beginning of the program and then control the running program.

At startup, the Injector analyzes the program code, makes the necessary changes to the code (this is explained in SummeRx docs), loads the requested parts of the code from the libraries and injects them into the code, etc. When the code is ready to be executed, the Injector runs it and handles exceptions at runtime. Thus, Injector is a dynamic part of SummeRx REXX program. The goal of the Injector is to preprocess (assemble) and start SummeRx REXX program. Using this technique, Injector provides modularity of the program, loose coupling for the provider-consumer model and other useful features.

Injector processing can be controlled by SummeRx REXX programmer with directives. The main directive is "++INC member" . This directive is used so that the injector inserts a member file into a stream instead of the current directive. Embedding directive is supported, i.e. an injected member can contain another SummeRx directive. The member itself can be a variable whose value is set in the configuration file of the application. This file is used to define both SummeRx and application global variables. For example, the names of SummeRx libraries are set in a configuration file of an application. It allows to search members only in certain libraries.

With the injector, programming in REXX has become much easier, because you no longer have to worry about passing stems to subroutines. Another big feature that was developed in SummeRx was the HEAP that implemented several modern OOP concepts on a basic level :

* Classes. Some elementary functionality is supported
* Garbage collector. Works in same thread of application
* Services. Elements of functional programming

Without going deep into the SummeRx HEAP architecture, it is important to emphasize that the use of HEAP primitives such as LIST, MAP or VARIABLE getters and setters makes programming in REXX much easier. The SummeRx HEAP is an implementation of a REXX stem that can be available in any SummeRx REXX program subroutine thanks to the Injector. In addition, the Injector can serialize/deserialize the HEAP between SummeRx REXX applications on manner of marshalling objects in JAVA.

SummeRx has a number of APIs offered with the framework :

* TN3270 applications
* TCPIP server
* Provider-Consumer pattern
* Multilayer messaging system
* Library and File System API
* Etc.

Below you can see some fragments of the code written with SummeRx:

1. Providing modularity via Injector. /\* REXX \*/

…. Template Start ….

….

…. Template End ….

**++INC SETBASE,ENV /\* Inject application configuration file \*/**

….

1. Using LIST class of HEAP.

/\* REXX \*/

…. Template Start ….

….

…. Template End ….

++INC SETBASE,ENV /\* Inject application configuration file \*/

stmxdata.NAME='EL01'

stmxdata.0 = 2

stmxdata.1 = "GL01"

stmxdata.2 = "GL02"

**/\* This will create an instance LIST01 of class LIST and add the list head element \*/**

**/\* EL01 with data from stem STMXDATA. \*/**

**parse value CLSMNGR("create","list",”LIST01”) with retcode retref RC\_EL elref1 .**

stmxdata.NAME="EL02"

stmxdata.0 = 2

stmxdata.1 = "itai"

stmxdata.2 = "yuval"

**/\* This will insert element EL02 to list LIST01 after element EL01 with data \*/**

**/\* from stem STMXDATA. \*/**

**parse value CLSMNGR("insert","list","LIST01",elref1) with retc elref2 .**

**/\* This will exchange elements EL01 and EL02 in list LIST01 \*/**

**/\* List class API has also functions ADD, DELETE, SORT, READNEXT, READPREV, etc. \*/**

**parse value CLSMNGR("exchange","list","LIST01",elref1,elref2)**

….

1. TCPIP-server. With a normal REXX it would take a long time to program. The result code will have low maintainability. With SummeRx it is a very short event-driven code as follows :

/\* REXX \*/

…. Template Start ….

….

…. Template End ….

++INC SETBASE,ENV /\* Inject application configuration file \*/

call TCPMNGR

/\* ================================================================ \*/

/\* Handle and message routines \*/

/\* ================================================================ \*/

/\* When Server got a client message \*/

/\* ================================================================ \*/

APPL\_RECEIVE : procedure expose rcvData readSock HEAP.

say "APPL:Got data "rcvData "through socket" readSock

return

/\* ================================================================ \*/

/\* When Server is ready to send a message to client \*/

/\* ================================================================ \*/

APPL\_SEND : procedure expose rcvData sendData writeSock HEAP.

sendData = "Hi,"word(rcvData,1)

say "APPL:Prepared "sendData "to send through socket" writeSock

return

/\* ================================================================ \*/

/\* When Server detected connection exception \*/

/\* ================================================================ \*/

APPL\_CLOSE : procedure expose recvData sendData readSock HEAP.

say "APPL:Connection exception detected on socket" readSock

return

Note that TCPMNGR includes socket event processing, message tracing, etc.

It is built to support standard server requirements: **request<->-response** and

**request<-> number of response** schemes.

Author : Gennady Lapidus

SailPoint Technologies

[summerexx@gmail.com](mailto:summerexx@gmail.com)

2020-12