pipeline |ˈpīpˌlīn|
noun
1 a long pipe, typically underground, for conveying oil, gas, etc., over long distances.
• a channel supplying goods or information: *the biggest heroin pipeline in history*.
2 Computing a linear sequence of specialized modules used for pipelining.
3 (in surfing) the hollow formed by the breaking of a large wave.
verb [with obj.]
1 convey (a substance) by a pipeline.
2 (often as adj. **pipelined**) Computing design or execute (a computer or instruction) using the technique of pipelining.
PHRASES
in the pipeline awaiting completion or processing; being developed: *new treatments are in the pipeline*.

Pipes for NetRexx

René Vincent Jansen, WarpStock Europe2019 Utrecht

May 18, 2019

A personal note

- I got acquainted with NetRexx through Pipes. I followed the NetRexx list for some time but never anything came from it. Playing with some homegrown stages for Pipes (re-)sparked my interest

- The **sqlselect**, **timestamp** and **xlate** stages in the Pipes product are the first NetRexx program code I wrote back in 1997

- The open sourcing of NetRexx prompted me to search for the remaining info on Pipes on the net, and to politely ask Ed Tomlinson (whom I have never met in person) to open source the pipes compiler

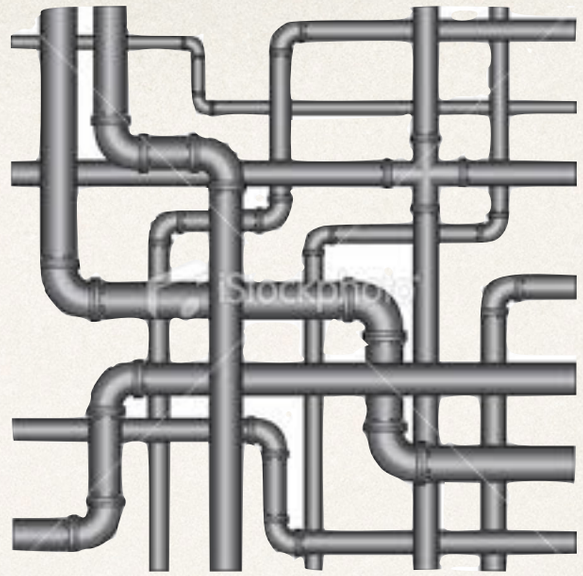
- He kindly granted his permission - this is why we are here today at this presentation

A caveat

- I am not a certified '*plumber*' - with regard to pipelines, I am an amateur. I only worked recently on z/VM - CMS.

- I used PIPE on TSO some 18 years ago - this was *after* being introduced to Pipes for NetRexx - but never for production.

- My sole intention here is to make it run and make it known - insure that people know about this very useful application of NetRexx on OS/2



What is it

Data Flow Programming - CMS - Hartmann Pipelines

Data Flow Programming

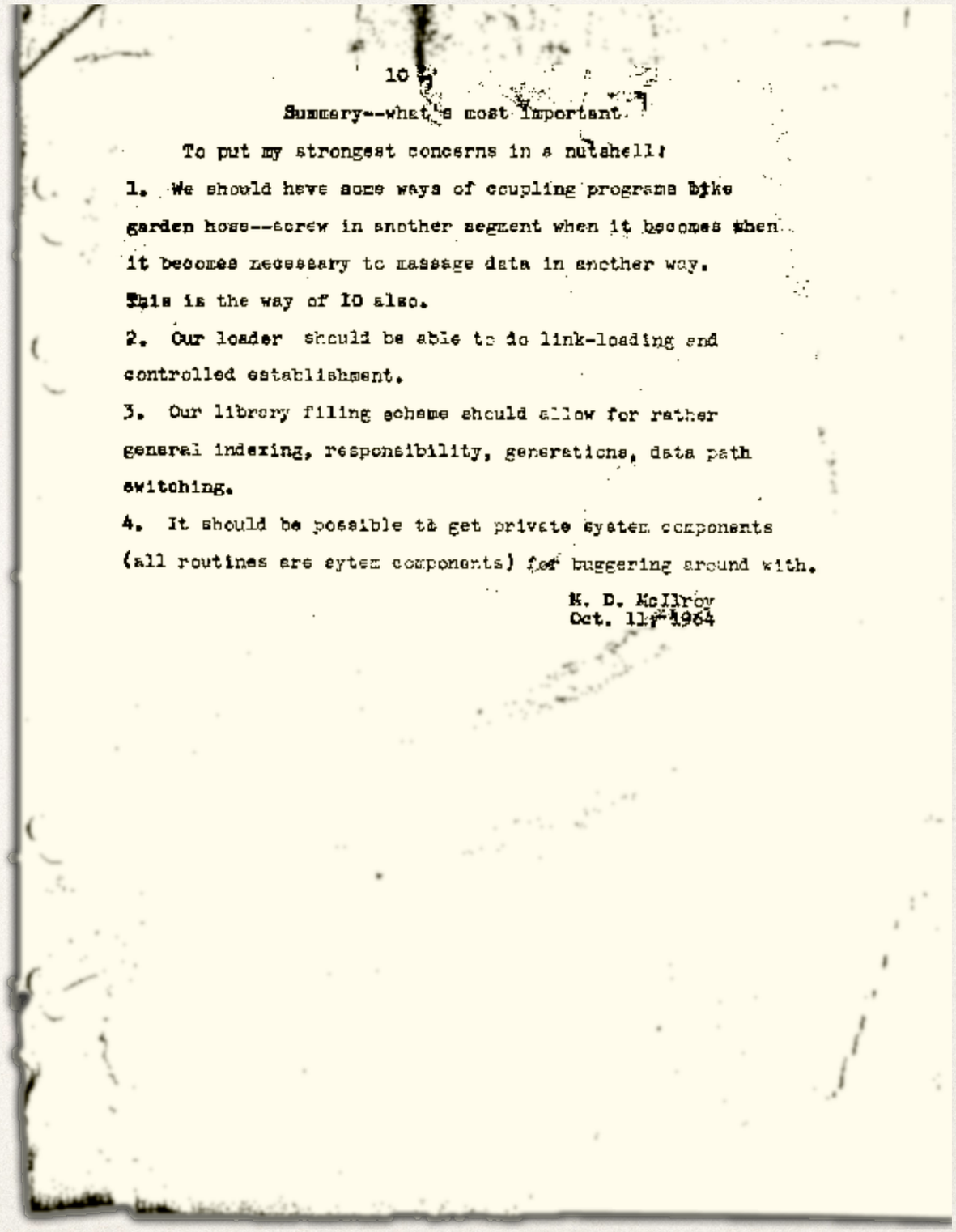
- Data flow programming is a suitable top level category for this subject

- In computer programming, data flow programming is a programming paradigm that models a program as a directed graph of the data flowing between operations, thus implementing data flow principles and architecture.

- The pipeline concept and the vertical-bar notation was invented by Douglas McIlroy, one of the authors of the early command shells, after he noticed that much of the time they were processing the output of one program as the input to another. His ideas were implemented in 1973 when Ken Thompson added pipes to the UNIX operating system. DOS, OS/2, Microsoft Windows, and BeOS also implemented the concept.

McIlroy memo

"Coupling programs like garden hose"



10

Summery--what's most important.

To put my strongest concerns in a nutshell:

1. We should have some ways of coupling programs like garden hose--screw in another segment when it becomes then it becomes necessary to massage data in another way. This is the way of IO also.

2. Our loader should be able to do link-loading and controlled establishment.

3. Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.

4. It should be possible to get private system components (all routines are sytem components) for buggering around with.

M. D. McIlroy
Oct. 11, 1964

A Unix pipe

- ls -l *.ls | grep | uniq

CMS

- Melinda Varian called Pipes "*the most significant addition to CMS since REXX*".

- CMS Users of Rexx see Pipes and Rexx as a symbiotic environment and tend to miss Pipes on other platforms. And complain about it.

- Unix pipes done a bit better, evolved over the 1980's.

- As presented during the 1991 International Rexx Language Symposium, "***How CMS Got Its Plumbing Fixed***", by John Poul Hartmann.

Hartmann Pipelines

Ceci n'est pas une pipe

- A **pipeline** consists of a collection of **stages**, joined together by stage **separators**. Stages can be written in a variety of languages, and are either filters that process data records or device drivers (sources and sinks) that read data into or out of the pipeline. Unlike other implementations of pipeline programming, Hartmann's design has multiple streams in and out of each stage and can interconnect them non-sequentially. Unlike many programming languages, pipelines have a very small amount of notation, limited to stage separators (typically "|"), pipeline separators (typically ";" or "?"), and label separators (":"). Due to common usage, the diskread stage is also known as < and diskwrite as >, however all stages have names that are words in or make some sense in English.

Differences from Unix pipes(wiki)

Filters may have multiple inputs and multiple outputs. For example, a selection filter can send the found records down one output pipe and the not found records down another.

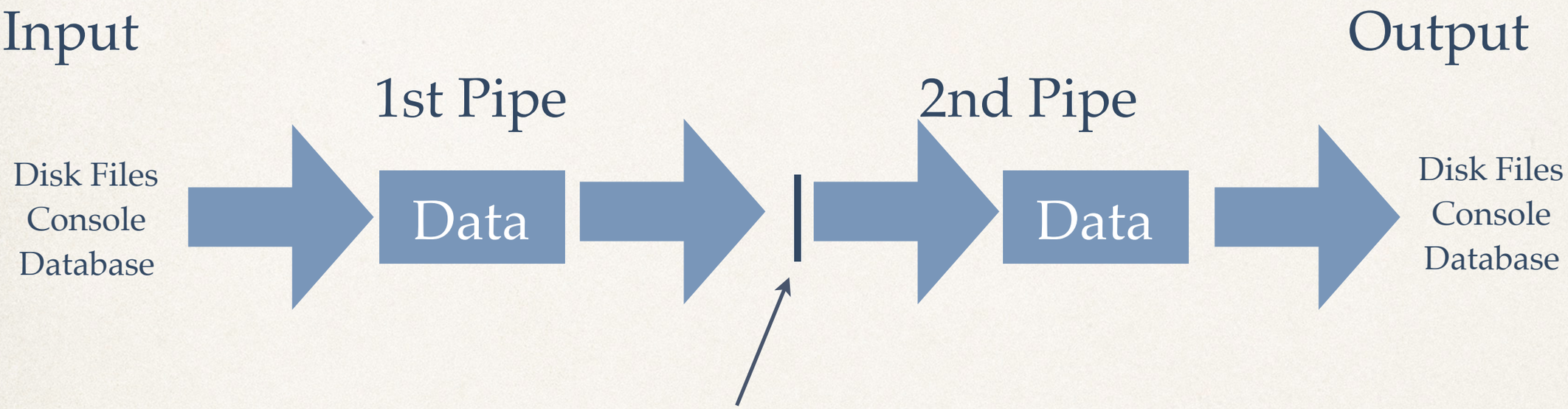
A linear notation for representing pipeline networks.

An interface that allows (Net)Rexx programs to act as stages.

A pacing strategy in the Pipeline supervisor that allows, for example, a stream to be split, say by a selection filter, and the records on the output legs to be processed by other filters, then merged by a join filter and have the record order preserved in result stream.

As implied by the previous item, data streams are (generally) not simply buffered and passed along to the next filter. The filters operate in parallel with input and output records handled by the Pipeline supervisor.

A Data Pipeline



! can replace | on Pipes and some national language versions

Device Drivers

Reading and writing disk files:

‘diskr‘ to read a file

‘diskw’ to create or replace a file

LITERAL:

A literal creates a record with the argument string and writes to a pipeline

- Combining input drivers:

This allows the programmer to create a file at one location and append, copy, or overwrite the file later in the pipeline.

The Console driver

- The console filter reads from the terminal and types on it; for example:

```
pipe console | console
```

CONSOLE can provide two functions:

Read input, when it is first in a pipeline specification

Type the input it gets, when it is not first

A device driver that writes to a device also writes the output to the pipeline.

Filters/Stages

- A filter is an application in a pipeline that takes its input from the stage to the left and passes its output to the stage to the right.

- The filters that are supplied with NJPipes have many general-use functions. They are also referred to as **Stages**.

- A function can be anything.

Buffer filters

- A filter that buffers a file reads all input records before writing output records.
- The SORT filter must buffer the file by the nature of its processing.
- Use BUFFER when a file must be buffered but not reordered.
- Examples:
- pipe diskr *inputfile* ! split ! sort unique ! console
- pipe console ! buffer ! stack

Discarding and keeping records

- Use TAKE and DROP to retain or discard a specified number of records from the beginning or end of the file.

- TAKE and DROP make it easy to select records based on their position in the file. Example: TAKE 5

- The DROP filter is the converse of TAKE, which allows you to delete the first or last n lines.

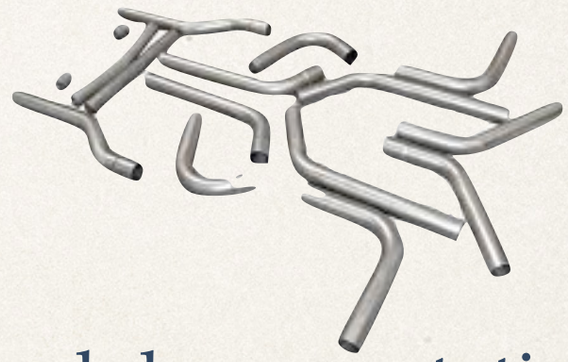
A simple example

```
pipe (xlatetest1)

   literal NetRexx is intended as a dialect of Rexx that can be efficient and portable. !
   xlate e Z t 64!
   console !
   compare: compare equal /OK/ notequal /BAD at \\c, \\b/ less /Less: \\s/ more /More: \\s/ !
   console ?
   literal NZdRZxx is indZndZd as a dialZcd of RZxx dhad can bZ ZfficiZnd and pordablZ. !
   compare:
```

The xlate stage documentation

```
/** xlate

                                                  <--------------------+
 >>--+-XLATE-----+--+----------------------+---+-------------------++---->
     +-TRANSlate-+  +-inputRange-----------+   +-+ default-table +-+
                    | <------------------+ |
                    +---(--inputRange--)-+-+

    <-----------------+
 >---+---------------++-------------------------------------------------><
     +-xrange--xrange-+

 default-table:
 +--+-UPper-------------------------+-----------------------------------+
    +-LOWer-------------------------+
    +-INput-------------------------+
 {  +-OUTput-------------------------+  }
 {  +-+-TO---+--+----------+--number-+  }
 {    +-FROM-+  +-CODEPAGE-+           }
 {                                     }
 { Not yet in njPipes                  }
```

```
-- implemented:
-- UPPER
-- LOWER
-- x y X Y (literal chars)
-- ) 40 (hex chars)
-- a-z = (range)
-- 61-7a = (hex range)
-- overriding of a range
-- E2A (EBCDIC to ASCII)
-- A2E (ASCII to EBCDIC)
-- column input range

-- not implemented:
-- WS word separator
-- translate table via input file
```



From the stage source - needs to be moved to formal documentation

What is NJPipes

The contents of the package - How to Install - Documentation

Date

The package

- NJPipes was originally released 1997-1999, and the source of the compiler was released in 2012. Pipes for NetRexx is integrated in the NetRexx package

- NetRexx is in a Git repository at SourceForfe. All which is needed to build and run, is in there.

- Build it by using make or the provided Windows batch files (thanks to Jean Louis Faucher for fixing these).

Compiler

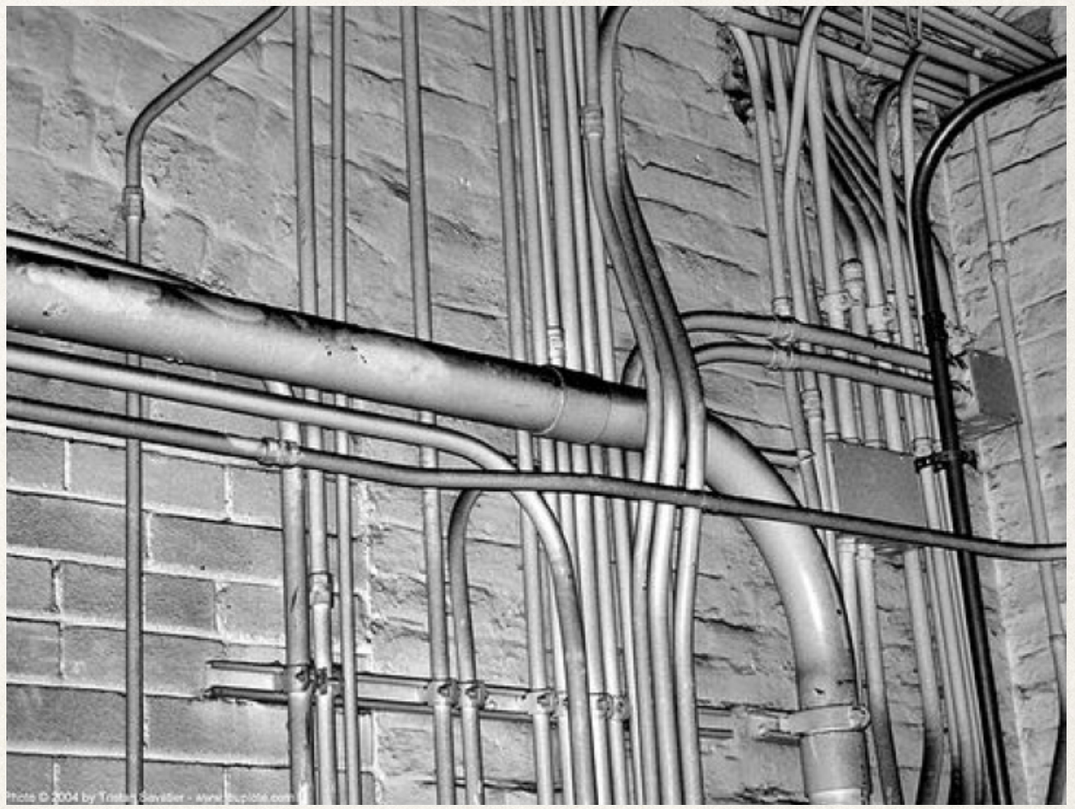
- The NJPipes compiler compiles - *and runs* - a program given a command line with device drivers and pipe stages

- It predates the interpreter in NetRexx (2000), some experimentation with interpreting stages is going on.

Use from NetRexx

- You can use a pipe in a NetRexx program if you call the file .njp and send it through the pipes compiler

- If you must, you can use pipes and stages from java programs.



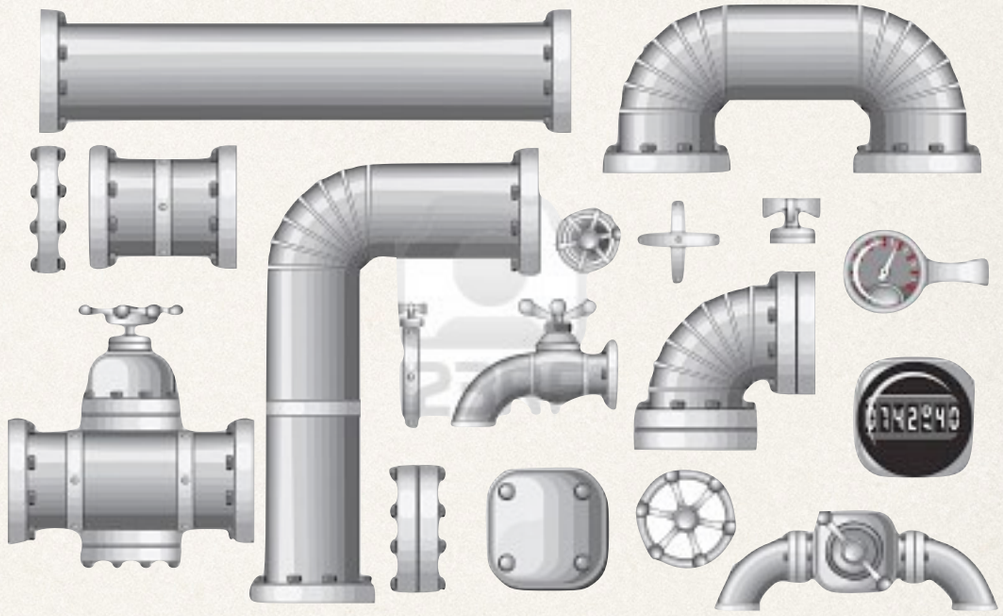
Stages

- A large number of stages is delivered with the product
- Many are modelled after their CMS Pipelines equivalent
- There, the CMS documentation might be used

Roll your own

- Also delivered is the capability to easily make your own pipe stages

- A simple template that can be fleshed out with your functionality



How to install

- The core classes for pipes and stages are in njpipesC.jar.

- This file may be used on the -cp option or added to your CLASSPATH.

Testing the installation

- To test your installation, you can type:

```
pipe "(test) literal
arg() | dup 999 | count
words | console"
```

To run the pipe type:

```
java test some words
```

The pipe should then output:

```
2000
```

Delivered Stages

- The following pages state all the stages that are currently delivered with NJPipes

- CMS in column 1 indicates compatibility with the CMS Pipes product implementation/ documentation

CMS	abbrev	
CMS	append	
CMS	array	
CMS	arraya	
CMS	arrayr	
CMS	arrayw	
CMS	between	
CMS	change	
CMS	casei	
CMS	chop	

	command	
	compare	
CMS	console	
CMS	copy	
CMS	count	
	dam	
	deblock	
CMS	deal	apl!
	dict	
	dicta	

	dictr	
	dictw	
	disk	
	diska	
	diskr	
	diskslow	
CMS	drop	
CMS	dup	
	elastic	
CMS	fanin	

	faninany	
CMS	fanout	
CMS	fblock	
	file	
	filea	
	filer	
	fileslow	
	filew	
CMS	frlabel	
	frtarget	

	gate	
	getfiles	
	getovers	
	getstems	
CMS	find	
	hash	
	hasha	
	hashr	
	hashw	
CMS	hole	

CMS	insert	
CMS	inside	
CMS	join	
CMS	joincont	
CMS	juxtapose	
CMS	literal	
CMS	locate	
CMS	lookup	
	not	

CMS	notinside	
CMS	nfind	
	over	
CMS	outside	
	pad	
CMS	pick	
CMS	reverse	
	rexx	
CMS	serialize	
	specs	

	prefix	
CMS	split	
	stem	
	stema	
	stemw	
	stemr	
	string	
CMS	strfind	
CMS	strnfind	
CMS	strfrlabel	

	sort	
	sortClass	
	sortRexx	
	sqlselect	
CMS	take	
	tcpclient	
	tcpdata	
	tcplisten	
CMS	timestamp	
CMS	tokenize	

CMS	tolabel	
	totarget	
	timer	
CMS	unique	
	var	
	vector	
	vectora	
	vectorr	
	vectorw	
CMS	xlate	

CMS	zone	

The 'rexx' stage

- Pipes for NetRexx moves objects. Many stages expect all objects to be of class rexx.
- The rexx stage modifier will convert objects to rexx if possible.
- ... ! rexx in change //xxx/ ! ... to insure inputs are rexx
- ... ! rexx out diskr ! ... to insure output are rexx
- ... ! rexx somestage ! ... both inputs and outputs are rexx

A Stage template

```
options binary
import org.netrexx.njpipes.pipes.
import org.netrexx.njpipes.stages.
class template1 extends stage

/* run a basic stage that has very little setup to do */
method run()

  /* insert objects that need to be reset every invocation here */
  rc = 0

  do -- to catch the terminating StageError

    /* setup code goes here
    *
    * if there are setup problems then
    *    signal StageError(11,'termplate1 had this error')
    *
    */

    /* body of the stage is here */

   loop forever
     object = peekto()       -- pass objects of any class
     output(object)
     readto()
   end

  catch e=StageError
  end

  rc = rc(e)     -- extract the rc from the StageError and update stage's rc

exit(rc*(rc<>12))
```

Using a pipe in a NetRexx program

```
import org.netrexx.njpipes.pipes.
import org.netrexx.njpipes.stages.

class testpipe extends Object
method testpipe(avar=Rexx)

F=Rexx 'abase'
T=Rexx 1

F[0]=5
F[1]=222
F[2]=3333
F[3]=1111
F[4]=55
F[5]=444

pipe (apipe stall 1000 )
stem F ! sort ! prefix literal {avar}! console ! stem T

loop i=1 to T[0]
say 'T['i']='T[i]
end

method main(a=String[]) static
testpipe(Rexx(a))
```

Debugging Pipes

- To find out what is happening in pipeline you have some tools. First, you can set a debug flag when you compile the pipe. The bits you set in the flag control what it does:

- 1 - Show all pipes starting
- 2 - Show all pipes ending
- 4 - Show all stages starting
- 8 - Show all stages stopping
- 16 - Show all Commit requests
- 32 - Show all Commit completions
- 64 - Show StageErrors raised via stage's Error(int,String) method. The
- stage class uses Error for all its StageError signals.
- 128 - Show the argument that each stage is receiving. Handy since
- shells have a habit of doing unexpected thing to arguments.
- (try: java findtext exit *.nrx vs java findtext "exit *.nrx")
-

Debugging Pipes II

- The second option is to use the invoke the dump() method in a stage This dumps the status of the pipe using the same format you see when a pipe deadlocks.

- Using dump() does not normally cause a pipe to terminate.

- Once in a while dump() will generate an exception. This happens since dump() does not use protect or synchronize so it does not stall.

Debugging Pipes III

- When all else fails, recompile using NetRexx **trace results**
- You will see everything that happens

What is there still to be done

After initial release, still work to be done

To Do:

- Organize the test cases into JUnit suites
- Complete the new documentation
- (John Poul Hartmann himself has promised he will write a foreword)
- Write more stages and fitting examples
- Make an installer
- You can be part of that!

Thank you for your attention

Q? rvjansen@xs4all.nl