independIT Integrative Technologies GmbH
Bergstraße 6
D–86529 Schrobenhausen

schedulix!focus

Controlling Workflows using Shell Scripts No Thanks!

Dieter Stubler Ronald Jeninga

November 25, 2016

Controlling Workflows using Shell Scripts - No, Thanks!

Introduction

As the title implies, this document is a critical consideration of the use of the UNIX shell (sh, ksh, bash, ...) for controlling the workflows of (sub-) processes.
The problems that arise and which are described here are, however, of a conceptual nature and can therefore also be applied to the use of other scripting languages âĂŃâĂŃ(Perl, Python, ...).
Under no circumstances are the languages mentioned here to be criticised across the board. Each of these languages has its advantages, and during his career the author himself has seen many successful projects completed using such script languages.

Workflow controlling of (sub-) processes

In any IT department there are tasks that have to be completed by executing multiple single processes. Such program workflows comprise just a few through to several hundred individual processes.
To handle such workflows, it has to be ensured that these single processes are coordinated, synchronised and completed in the right order. This requires a workflow controller.
In practice, the UNIX shell or other script languages âĂŃâĂŃare frequently used to implement such workflows and control them.

Everything is simple at the beginning

Things are always easy at the initial phase of a project. The number of workflows is small, the complexity is minimal, the volume of data is manageable, and workflow controlling is not (yet) an issue. Pretty much every workflow can be regulated quickly, pragmatically and cheaply with just a few lines of shell script.
And precisely this situation is laden with risks because these solutions, in conjunction with increasing complexity and heavier demands being placed on the performance and reliability of the overall system, tie down considerable development resources and in the medium term make it impossible to reliably run the workflows at a viable cost.
From the perspective at management level, the fatal aspect here is that the workflow control and monitoring costs are initially so negligible, there is not even separate budget for them. That's why the always insidious increase in these costs stays unnoticed by senior management for an extremely long time (idle performance!). When the issue is finally recognised as being a problem, huge sums have already been sunk into the development of control scripts and the necessary workarounds and frameworks.

To avoid losing this investment, and through fear of the expense of switching the workflow control mechanism to a suitable scheduling system, companies often attempt to stick with their existing work methods. The costs continue to rise, more and more workforce resources are tied down, and moving away from this inefficient work method becomes increasingly expensive.
In this document, we want to raise awareness about this problem, and demonstrate that the early deployment of a suitable scheduling system can prevent the dilemma described here from arising in the first place, and that costs can be reduced right from the very beginning and resources can be freed up for solving the actual tasks in hand.

A small example

We want to demonstrate the development of an elementary workflow using an extremely simple example. In doing so, we assume that for a simple workflow two programs (P1 and P2) are to be executed one after the other.
We begin by writing the following sh script:

```
#!/bin/sh

P1
P2
```

That's really easy, isn't it?

Troubleshooting

Errors are not handled in the above example. To make sure that P2 does not process the wrong data, P2 must be prevented from starting if P1 returns an error. Since this has already caused trouble several times, we have to incorporate an error handling routine into our script. That looks like this:

```
#!/bin/sh

P1
RET=$?
if [ $RET -ne 0 ]
then
        echo "Error $RET in P1 !"
        exit $RET
fi
P2
RET=$?
if [ $RET -ne 0 ]
then

        echo "Error $RET in P2 !"
        exit $RET
fi
```

As we can see, a few lines have already been added. These have to be tested because an error that remains unremedied can have fatal consequences. Error handling itself is implemented in a slightly varying manner by different developers. Comprehensibility and maintainability diminish and costs rise.

Restart

In our example, P1 has a runtime of 3 hours and P2 often returns an error caused by a resource bottleneck in the temporary memory. Restarting the script would also unnecessarily cause P1 to be run again, resulting in an extra repair time of 3 hours. In many environments, the developer would now copy the script, comment out the call for P1, and run the copy of the script. This is extremely work-intensive and carries with it a high risk of errors occurring.
Since this is not acceptable, the script needs something like a memory. Then it looks like this:

```
#!/bin/sh

STEP=`cat stepfile`

if [ $STEP -eq "0" ]
then
        P1
        RET=$?
        if [ $RET -ne 0 ]
        then
                echo "Error $RET in P1 !"
                exit $RET
        fi
        echo "1" > stepfile
fi

if [ $STEP -eq "1" ]
then
        P2
        RET=$?
        if [ $RET -ne 0 ]
        then
                echo "Error $RET in P2 !"
                exit $RET
        fi
        echo "2" > stepfile
fi

echo "0" > stepfile
```

This is a quick way which the developer might decide to use. However, the aforementioned solution harbours numerous problems. The 'stepfile' has to be initialised before it is run for the first time and possibly after the processing has been aborted. The error handling routine for reading and writing the progress is still

lacking entirely. With this solution, the script can always only be run once at any given time. To make the script more productive requires the investment of considerably more development time.
At this point, we'll dispense with more sample code for our example because several hundred lines of code would quickly become necessary and we can obviously save ourselves the time writing them.

What else?

The example above only deals with the initial simple problems of controlling a workflow using a script. To ensure stable and smooth operation requires the following additional functions (among others):

- Monitoring and operator intervention.
- Transfer of control information.
- Parallelism capability.
- Distributed execution.
- Resource controls.

Implementing these and other functions in the workflow control routine requires spending considerable time and expense on development and maintenance.
A lack of any of these features will be paid for by higher operating costs.

Monitoring and operator intervention

Before a stable operation can be ensured, it must be possible to monitor all the processes involved. This requires that all the processes log their progress ('stepfile' in our example). These logs have to be collected and prepared to acquire a status overview of the ongoing processes. This necessitates a repository containing state information about ongoing processes. Incidentally, to be quickly informed about any problems that arise, a notification system also needs to be developed that can send messages via e-mail, text or other channels. But just developing these functions can turn into a small-scale project which can quickly involve the investment of man-days in double figures.
If any errors or problems arise, the operator must be capable of intervening in ongoing processes. He must be able to restart, pause, stop, skip, etc. workflows or processes. Achieving this in anything remotely representing a convenient manner will also involve enormous development costs.

Transferring control information

It is often necessary to hand over information (timestamp, file names, ...) from one process to another downstream process. In a script-based workflow control routine, this is typically implemented using files that are written by one process and then read by its successor. However, this solution again presents many disadvantages. Considerable development work is necessary to re-implement the transfer of information at the latest when one of the participating processes running on another machine in the network has to be run.

Parallelism capability

The development of scripts moves into a new dimension if sub-processes can be run consecutively. Processes have to be started 'in the background' and there has to be a pause at certain points of the workflow until multiple parallel processes have been completed. This requires a high degree of technical know-how on the part of the developer and, including the error handling, restarting, monitoring and operation, is by no means a trivial task.

Distributed execution

If the sub-processes of a workflow are to be run on different computers, processes have to be started from the scripts on these computers. Again, this is no easy task because the pitfalls of remote commands (rsh, rcp, ...) need to be overcome with regard to monitoring and identifying errors.
And we don't even want to begin talking here about the potential security risks, such as the encoding of passwords.

Resource controls

The available system resources are always limited. If too many resources are needed at a given time, this results in a lower data throughput and errors occur more frequently due to resource bottlenecks. It therefore has to be ensured that processes only start if sufficient resources are available. It is practically impossible to realise this with justifiable effort using a script-based workflow control mechanism.
So far, we have only touched upon a few of the key aspects of workflow controlling and the time and effort involved in scripting. The lack of an appropriate scheduling system will result in significant ongoing operating costs being incurred. Attempting to lower these costs by improving the script infrastructure will involve much more development and associated maintenance work.

schedulix as an alternative solution and way out

With its schedulix Scheduling System, independIT offers an alternative solution and a way out of the script trap.

The system provides all the requisite functions for modelling even large-scale and complex workflows without the need to implement parts of the workflow control mechanism in the sub-processes themselves.
The time and cost spent on development, maintenance and operation are cut drastically by deploying the schedulix Scheduling System.
Not only that, but the operation of the data warehouse becomes more stable, less prone to errors and more reliable. Recovery times can be significantly reduced.

Closing remarks

Controlling workflows on a script basis necessitates inordinately high development, maintenance and operating costs. An optimised, transparent and efficient operation cannot be achieved with script-based workflow controlling.
For this reason, we recommend utilising a suitable scheduling system right from a very early stage. The sooner a switch is made to such a system in an ongoing project, the lower the investment loss, and it is consequently less expensive to make the changeover. If scripts are being used to control workflows in your environment, you should strive to switch to a scheduling system-based solution as quickly as possible.
The independIT schedulix Scheduling System features all the functions required for developing, running, monitoring and operating complex workflows, and thus minimises the expense of developing and maintaining process workflows. For just a fraction of the cost of a scripting solution, the schedulix Scheduling System will help you to set up and operate a sustainably stable and reliable IT environment.
Act now!