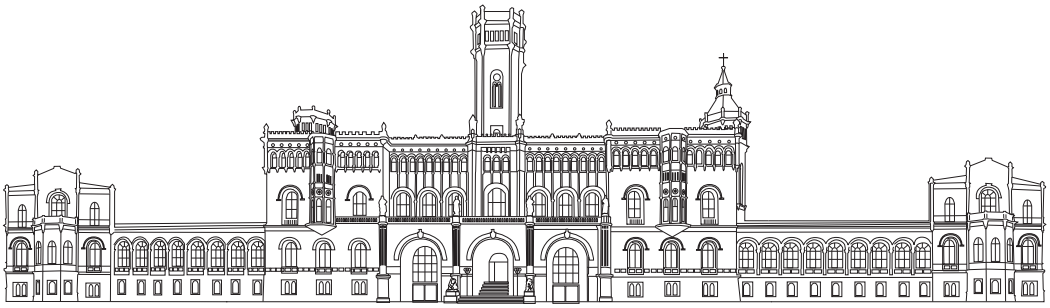

LEIBNIZ UNIVERSITÄT HANNOVER

FAKULTÄT FÜR ELEKTROTECHNIK UND INFORMATIK
INSTITUT FÜR KOMMUNIKATIONSTECHNIK

SSHLauncher

Technical Report

ZDRAVKO BOZAKOV

01. Mai 2009

[January 11, 2011 at 16:42]

CONTENTS

1 SSHLAUNCHER OVERVIEW

Simulation is a widely used paradigm for verifying existing theories as well as novel approaches in communication networks. In order to ensure the validity and credibility of simulation results, it is crucial that sufficiently large numbers of simulation runs are performed and statistics and corresponding confidence intervals are presented. Unfortunately, the generation of sufficiently large amounts of data can often be painstaking. In fact, the widespread credibility problem of simulation studies has been analyzed in [5] and [3].

Setups which rely on a number of different tools are typically cumbersome to configure and difficult to automate. In cases where a set of tools must be executed in a distributed environment, the task of efficiently performing large numbers of experiments becomes even more difficult.

Towards this end, the *SSHLauncher* software aims to facilitate the automated execution of experiments in distributed environments. Moreover, the software allows users to execute large sets of experiments and study the effects of parameter variation, with minimal user intervention. The repetition of experiments becomes a matter of executing a single command, allowing researchers to concentrate on research tasks. In the course of our research we did not find existing tools which adequately address these goals.

Each experiment is defined by a clear configuration script. As a side effect, users are encouraged to generate documentation for their work. Moreover, the experiment configuration file can be made available to the community leading to easily reproducible results. Faults in the experimental setup can be identified and addressed more quickly. The importance of reproducible research has been pointed out in [7, 6].

2 SYSTEM SETUP

SSHLauncher enables users to automate the execution of a set of commands on different hosts. A typical setup is depicted in Fig. 1. The hosts, corresponding commands and the dependencies which govern the execution order are specified in a configuration file stored on the control machine. As a result, after an initial configuration has been specified, complex experiments may be repeated an arbitrary number of times requiring almost no interaction from the user. The sole prerequisite is that all hosts allow remote SSH logins from the control machine. This is the case for almost all UNIX based operating systems and, using Cygwin, also under Windows. Additionally, a Python installation must be available on the control machine. Note that for experiments sensitive to network load it is advisable that separate data and control interfaces are present in the experiment setup, in order to minimize interference with ongoing measurements.

The Emulab [9] and PlanetLab [2] testbeds are examples of highly suitable environments for the use of *SSHLauncher*.

SSHLauncher utilizes the *pexpect* and *pxssh* Python modules written by Noah Spurrier [8], which are also included in our distribution.

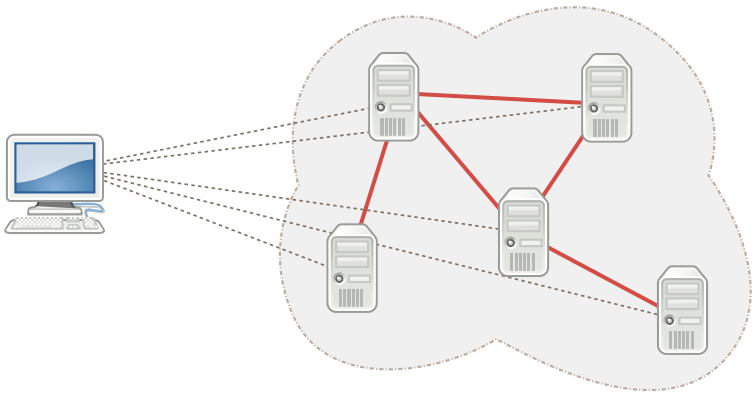

Figure 1: Typical SSHLauncher setup. Dashed lines represent control connections.

3 USAGE

3.1 PROGRAM CONCEPT

The idea behind *SSHLauncher* is to automate the process of logging into remote hosts and executing a set of software tools. To achieve this the experiment commands are split into blocks and stored in a configuration file. Each configuration file block is associated with a specific SSH session. Similarly to the UNIX program `expect`, dependencies based on output strings of individual commands may be specified. In other words, each block of commands may contain numerous constraints as to when they can be executed. An analogy from project management are tasks in a Gantt chart, where certain tasks cannot begin before one or more other tasks have been completed.

The *SSHLauncher* batch execution is initiated using the following command:

```
$ sshlauncher [-d] CONFIGURATION
```

where `<config>` is a mandatory configuration file containing a series of hosts, commands and dependencies governing the execution order. The configuration format will be described in detail in the following chapters. The program establishes an SSH connection to all hosts specified in the configuration file and executes the specified commands, taking into account all given dependencies. As soon as the execution of a specific configuration block is finished, i.e. the bash prompt is reached on the destination host, the SSH session is terminated. The *SSHLauncher* terminates, when all configuration blocks have been evaluated.

The optional parameter `-d` enables the debugging mode, causing *SSHLauncher* to generate log-files containing the terminal output for every opened SSH connection. A log file is created for each configuration file section in the current working directory. Additionally, the program verbosity is increased.

4 CONFIGURATION FILE SETUP

4.1 NOTATION

Throughout this document <> brackets will be used to denote an arbitrary text string. Section tags not placed in brackets must be written exactly as depicted. If not mentioned otherwise, square brackets are to be treated as the literal characters [and].

The *SSHLauncher* configuration file consists of a series of so-called blocks or sections which specify the commands to be executed on the destination host, as well as the order in which these are to be run. The file format is based on the windows *.INI* file convention. Each section is identified by a unique section id and is associated with a single SSH connection. The corresponding SSH session is initiated when all block dependencies have been met, and is terminated when all block commands have finished executing. The id tag may contain any sequence of letters and digits and must be enclosed in square brackets. The section id is followed by a number of parameters identified by special tags. The tags include a user-name, destination host name and the commands to be executed. Furthermore, a password can be specified if SSH key authentication is not configured. The format for a configuration file section is as follows:

```
# <comment>
[<id>]
user: <name>
password: <password>
host: <hostname>
command: <bash_commands>
```

The section tags are case-sensitive however, their order is not relevant. Moreover, the order of the file sections may also be arbitrary and does not effect the execution order. The number of sections in a configuration file is only limited by the maximum number of active SSH connections allowed by the system. Lines beginning with # are treated as comments.

In addition to the tags described above, sections may contain the *after* tag. This tag implies, that the current section command should only be executed after a specific string has been issued by a command of another section. It is possible to include several id/string pairs, separated by commas. In this case the section command is executed after all strings have been matched. The *after* tag uses the python language dictionary notation:

```
after: {'<id1>':'<string1>', ... ,'<idN>':'<stringN>'}
```

4.2 DEFAULTS AND VARIABLES

In addition to the sections described above, which are always associated with an SSH connection, a special *default* section can be specified in the configuration file. This section contains default tag/value pairs which are appended to all other section in the file. Tags also defined in a section override tags specified in the default section. Furthermore, it is possible to define variables which are accessible from within the sections. The format of the default section is described below:

```
[DEFAULT]
<varname1> = <value1>
<varname2> = <value2>
#
user: <name>
password: <password>
```

Variables can be useful as a shorthand for frequently used pathnames or log filenames. To substitute a variable into any other part of the configuration file formatted as a string the following notation must be used: `%(<varname>)s`. For more details on variable usage, please refer to the examples chapter 6 of this document.

4.3 EXTERNAL VARIABLES

It is possible to define special variables in the Bash shell, which are imported and substituted into the configuration file. This feature can be useful when performing large numbers of experiments, where only a few parameters change in every run. The Bash variables are declared as usual, however their name must be prefixed with `SL_`.

A typical Bash script that invokes *SSHLauncher* with a varying parameter, in this case packets per second, is depicted below:

```
#!/bin/bash
for ((i=1;i<=24;i++)); do
    echo "==== running test nr. $i ===="
    # calculate packets per second
    let SL_PPS=${SL_I}*85
    export SL_PPS
    export SL_I=$i
    sshlauncher.py spruce.config
    scp zbozakov@node0.Dumbbell.Experiment.emulab.net:logs/*bz2 ./
done
```

In the example above the variables `SL_I` and `SL_PPS` are available from within the configuration file `spruce.config` and may be used as follows:

```
...
command: ITGSend -a node4 -t 600000 -C 2040 -c %(SL_PPS)s -rp 8999
...
```

5 NOTES AND TIPS

Within a configuration section, it is legal to concatenate several Bash commands using standard Bash operators such as `;` or `&&`. These commands are then executed sequentially using a single SSH connection. On the other hand, execution speed can be increased if commands are run in parallel using separate SSH connections. In order to keep the number of simultaneously opened SSH sessions to a minimum, a connection is only established when all strings in the *after* tag have been matched.

Certain programs do not produce any text output. If other SSH section commands depend on the termination of such a program, the user can manually generate a string after the program completes using the *echo* command. This works reliably using the Bash shell.

```
command: <bash_command>; echo 'finished command'
```

For tools which do not have a built-in logging support, it is often desirable to redirect the output to a file. At the same time the program's output should be available to *SSHLauncher* in order to trigger the execution of other command blocks. The Unix command *tee* can be useful for these cases.

```
command: %(bindir)s/iperf -s | tee -i -a
%(logdir)s/%(SL_I)s_iperf_rcv.log
```

A common source of problems is that some programs pipe their output to *stderr*, making it difficult for *SSHLauncher* to pick up the output. To circumvent this, Bash I/O redirection may be used, e.g. `2>&1` redirects *stderr* output to *stdout*.

```
command: %(bindir)s/pathchirp_snd 2>&1 | tee -a -i %(logdir)s/snd.log
```

To avoid polluting the Bash history of the destination hosts, history saving is disabled.

It should be noted that the user must take care to avoid circular references in the configuration file, which would cause the scripts to wait indefinitely.

6 SAMPLE CONFIGURATION

In this section a configuration file from a real-world scenario will be outlined. A simple dumbbell setup in Emulab will be used to start two sender and two receiver instances of the rude/crude traffic generator [4]. The topology is depicted in 2. The first flow will be sent from node0 to node5 and the second one from node1 to node4.

First we will define some default variables containing the tool paths. Because the username is the same for all SSH connections it is also included here. In our example SSH-key authentication is configured, so there is no need to specify a password.

```
[DEFAULT]
log = /proj/Experiment/exp/results/active
bin  = /proj/Experiment/exp/wifi-simple/tools/rude_0.70/bin
home = /users/zbozakov

user: zbozakov
```

Next, we will do some cleaning up, i.e. remove all log files which might have been left over from previous runs. As this section has no *after* tag, it will be executed right at the beginning. Note that because the `rm` command usually does not produce any output, a string is manually printed after the file deletion.

```
#### cleanup dirs ####
[cleanup]
host: node3.Dumbbell.Experiment.emulab.net
command: rm %(log)s/*bz2; rm %(log)s/*log; echo 'logs deleted'
```

In the next two sections the receivers are started on nodes 4 and 5. The *after* tag ensures that the cleanup section has completed before the connections are established. Moreover, we make use of the variables defined in the default section.

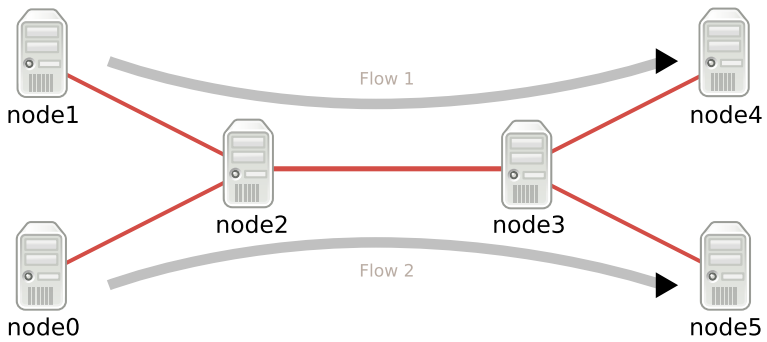

Figure 2: Dumbell topology

```
### start receivers ####
## probe traffic receiver
[prcv]
host: node4.Dumbbell.Experiment.emulab.net
command: killall crude; %(bin)s/crude -P 1 -p 10001 -l
%(log)s/p_crude.bin.log
after: {'cleanup':'logs deleted'}

## cross traffic receiver
[xrcv]
host: node5.Dumbbell.Experiment.emulab.net
command: killall crude; %(bin)s/crude -P 1 -p 10001 -l
%(log)s/x_crude.bin.log
after: {'cleanup':'logs deleted'}
```

After the logs have been deleted and both receivers are running, the senders can be started. Because the receiver (*crude*) prints its version information immediately after start up, we can use the string `'crude version'` for matching. To ensure that old instances of the sender (*rude*) will not interfere with the current run, these are killed first. Both senders run for a predefined time and we emit a string to signal that they are finished.

```
#### start senders ####
## probe traffic sender
[psnd]
host: node1.Dumbbell.Experiment.emulab.net
command: killall rude; %(bin)s/rude -s %(home)s/prb.cfg >
%(log)s/psnd.log; echo end
after: {'cleanup':'logs deleted', 'xrcv':'crude version',
'prcv':'crude version'}

## cross traffic sender
[xsnd]
host: node0.Dumbbell.Experiment.emulab.net
command: killall rude; %(bin)s/rude -s %(home)s/trc.cfg >
%(log)s/xsnd.log; echo end
after: {'cleanup':'logs deleted', 'xrcv':'crude version',
'prcv':'crude version'}
```

Finally, we include two more sections which kill the receivers and compress the generated log files after both senders have terminated, i.e. the string `end` has been issued by the senders.

```
#### terminate receivers and compress logs ####
## probe traffic receiver
[prcv_end]
host: node4.Dumbbell.Experiment.emulab.net
command: killall crude; bzip2 %(log)s/p*.log
after: {'psnd':'end'}

## cross traffic receiver
[xrcv_end]
host: node5.Dumbbell.Experiment.emulab.net
command: killall crude; bzip2 %(log)s/ar*.log
after: {'xsnd':'end'}
```

7

SUMMARY

We presented *SSHLauncher*, a tool which facilitates the execution of multiple commands with dependencies in a distributed environment. Using an intuitive configuration file syntax, large sets of complex experiments can be performed with minimal user interaction. *SSHLauncher* was born out of necessity and was heavily employed for generating data in [1]. We hope that other users will also find the software useful.

The current version of the *SSHLauncher* software together with this documentation can be found at: http://www.ikt.uni-hannover.de/software.html

ACKNOWLEDGEMENTS

This work has been funded by an Emmy Noether grant of the German Research Foundation.

BIBLIOGRAPHY

[1] Michael Bredel and Markus Fidler. A Measurement Study of Bandwidth Estimation in 802.11g Wireless LANs using the DCF. In *Proceedings of IFIP Networking*, number 4982 in LNCS, pages 314–325. Springer, May 2008.

[2] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: An Overlay Testbed for Broad-coverage Services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/956993.956995.

[3] Stuart Kurkowski, Tracy Camp, and Michael Colagrosso. Manet simulation studies: the incredibles. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(4):50–61, 2005. ISSN 1559-1662. doi: http://doi.acm.org/10.1145/1096166.1096174.

[4] Juha Laine, Sampo Saaristo, and Rui Prior. Real-time UDP Data Emitter & Collector for RUDE, 2008. URL http://rude.sourceforge.net/. http://rude.sourceforge.net/.

[5] K. Pawlikowski, H.-D.J. Jeong, and J.-S.R. Lee. On Credibility of Simulation Studies of Telecommunication Networks. *Communications Magazine, IEEE*, 40(1):132–139, Jan 2002. ISSN 0163-6804. doi: 10.1109/35.978060.

[6] Robert Gentleman and Duncan Temple Lang. Statistical Analyses and Reproducible Research. *Journal of Computational and Graphical Statistics*, 16(1), March 2007.

[7] Matthias Schwab, Martin Karrenbach, and Jon Claerbout. Making scientific computations reproducible. *Computing in Science and Engineering*, 02(6):61–67, 2000. ISSN 1521-9615. doi: http://doi.ieeecomputersociety.org/10.1109/5992.881708.

[8] Noah Spurrier. Pexpect - Pure Python Expect-like module. http://www.noah.org/wiki/Pexpect, 2008. URL http://www.noah.org/wiki/Pexpect.

[9] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 255–270, Boston, MA, December 2002.