A Networkless Data Exchange and Control Mechanism for Virtual Testbed Devices

Tim Gerhard(✉), Dennis Schwerdel, and Paul Müller

Integrated Communication Systems Lab, University of Kaiserslautern, Kaiserslautern, Germany
{t_gerhard10,schwerdel,pmueller}@informatik.uni-kl.de

Abstract. Virtualization has become a key component of network testbeds. However, transmitting data or commands to the test nodes is still either a complicated task or makes use of the nodes' network interfaces, which may interfere with the experiment itself. This paper creates a model for the typical lifecycle of experiment nodes, and proposes a mechanism for networkless node control for virtual nodes in such a typical experiment lifecycle which has been implemented in an existing testbed environment.

Keywords: Testbed · Control Interface · Node Control

1 Introduction

Network research is becoming more important since the Internet and other computer networks have a growing influence on the world. For this area of research, network testbeds are a crucial tool for experimentation. These testbeds usually offer a number of devices distributed over the globe with certain connection configurations between them. The experimenters' influence on this setup and its variables depends mainly on the testbed's architecture.

An important aspect for the usage of a testbed is how the network devices can be controlled. For large experiments which have many network nodes it is not feasible to control every device by hand. Thus, the experimenter needs to have a controlling interface which can be automated, i.e. scripted. Many testbeds (such as PlanetLab [3] or EmuLab [2,7]) use the devices' networking capabilities to provide such an interface. Automation frontends for these testbeds like gush [1] also need a network connection to the devices.

However, in a networking testbed, a network interface (especially one connected to the Internet) may not be a good solution to the problem of controlling a device. There are several disadvantages when choosing this control method which have to be accounted for in the experiment design.

configuration. Depending on how node control is realized, there is either an additional network interface on every device or one of the interfaces which is being used in the experiment is also used for control. In the first case,

V.C.M. Leung et al. (Eds.): TridentCom 2014, LNICST 137, pp. 14–22, 2014.
DOI: 10.1007/978-3-319-13326-3_2

the experiment must be configured never to use the additional interface, even when routing over this interface would make more sense than routing over another one. In the second case, this interface is forced to support the traditional protocol stack including TCP/IP.

traffic. There may be uncontrolled traffic coming from the outside network to the experiment. This may affect measurements as this additional traffic uses bandwidth, may cause additional latency or interfere with the experiments in other ways.

connectivity. There may be experiments which may not be connected to the Internet for several reasons. For example, you cannot run a malware analysis while connected to the Internet without endangering the Internet (Such an experiment has been done on ToMaTo, using VNC as the node control method [5]).

Testbeds can provide a networkless control interface for these kinds of experiments, which will be shown in this paper. In Section 2, a model for automated node control will be developed. Section 3 describes how these operations can be realized in a host-guest system, section 4 introduces the actual implementation in the Topology Management Tool (ToMaTo [4,6]) and section 5 concludes this paper.

2 Requirements for Automated Control

After creating devices, the experimenter will usually install software on it (1), configure it (2), run the experiment (3) and then collect the resulting data (4). Step 1 consists of transmitting files to the device and then execute the installation routine. Step 2 also consists of running commands and maybe uploading some configuration files to the device. Step 3 can be initiated by a command, and step 4 is a file download from the device. Every additional interaction can also be possible through file transmission or sending commands.

For an automated control, one must be able to wait for a command to finish before continuing with the next step. Therefore, an automated control interface only needs these three operations: transmitting files between the controlling and the controlled device, executing commands or scripts on the controlled device and monitoring the progress of this execution.

Instead of allowing to directly execute a defined command, the controlled device can be configured to automatically execute a script identified by a certain file name after such a script has been uploaded. Uploads and downloads are done through archives, where the archive will be extracted to a certain directory after an upload, and the archive will be created from this directory again for download. For the purpose of describing, archive and directory can be viewed as equivalents.

A system which provides these three operations (upload and execute, query execution status, download) for its devices to its users without using the target device's network interfaces provides *Remote Execution and Transfer of Files for Vitual computers* (RexTFV).

3 Communication Between Host and Guest

This paper will focus on the interface between host and guest. It does not describe how the host is controlled by the user, but it is assumed that the additional commands can be integrated into the testbed's architecture.

Storage is a resource which is shared between host and guest. In general, the guest can access a part of the host's storage. This fact can be used to emulate a shared directory, which can then be used to provide the operations described in Section 2.

The *network-less Execution and Transfer Protocol* (nlXTP) uses such a shared directory to provide these operations between host and guest systems. The term *network-less* means that it does not make use of network interfaces.

RexTFV has been designed to work for virtual devices, but it can be used in any scenario where the controlling node can access the controlled node's storage.

3.1 Shared Directory

Virtualization systems can be categorized into container-based or full virtualization, which are completely different approaches to the problem of virtualizing computer systems. Thus, there are fundamental differences in the realization of the shared directory.

As will be shown in the next section, the shared directory has to provide the following:

- upload of an archive,
- download of an archive, and
- a frequent, scheduled reading of a certain file (the status file) by the host, written by the guest.

It is assumed that the user does not execute the upload and download operations while the guest is still working on the files, given the fact that the user knows when operations are running. Thus, only the scheduled reading of the status file has to cover possible inconsistency.

Container-Based Virtualization. Cantainer-based virtual machines (such as OpenVZ or VServer) aim at creating a different runtime environment, while host and guest system still share one kernel, including drivers. This means that the virtual machine is integrated into the host's scheduler and file system. In fact, the guest's root directory is simply a certain directory in the host's file system. Since nlXTP requires full control over the shared directory, this shared directory must be an otherwise unused subdirectory of the guest's file system.

Both host and guest can access the directory at any time, reading or writing. The only occurence of inconsistency may happen if the host reads a file which is at this point of time being written by the guest. To prevent this, the usual ways of preventing simultanous access to one file by multiple processes can be used. Alternatively, the file can be secured by a checksum.

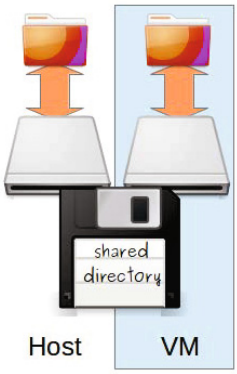

Fig. 1. The virtual disk can be mounted in both systems simultaneously

Full Virtualization. In full virtualization systems (like KVM/QEMU or VirtualBox), such a shared file system can be realized by a virtual disk, which can be accessed by both the host system and the guest system (see figure 1). This disk needs to have a file system which is supported by both systems (in many cases vFAT is suitable).

To avoid an inconsistent file system, host and guest must never write to this disk at the same time, or before the cache of the other system has been written back. Since it must be assumed that the disk is always mounted by the guest system while it is turned on, the host system must only write on the disk while the guest system is shut down. This means that archive uploads are unavailable while the guest system is running.

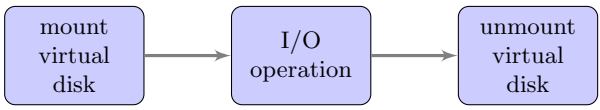

Fig. 2. Access Sequence when the host performs an I/O operation on the shared directory

However, the host system can still read the disk while the guest system is turned on. To lower the probability of an inconsistent file system while reading, the host system only mounts the disk right before reading or writing, and unmounts it right after the reading (see figure 2). Assuming a write-through caching strategy by the guest, and given the assumption from above (the user does not start a download while the the guest is still writing on the files), the guest writing in the status file while the host is reading it remains the only chance of inconsistency.

There may be three kinds of effects: (1) The file does not exist, (2) The file does not fit into the boundaries described in the disk's file table or (3) the

file is being changed by the guest while the host is reading it (thus, the data is corrupted). To avoid all these errors, the guest secures the file content by a checksum. In case 1, the inconsistency can be detected directly (assuming that the file must exist; if it doesn't, the whole operation is pointless). In case 2, the checksum does not exist (or case 3 applies, depending on the implementation) and in case 3, the checksum validation will fail. If inconsistency is detected, the reading can be repeated after a short interval of time: just enough so the guest can finish the operation on the file.

3.2 Operations

NlXTP provides operations according to RexTFV in section 2. These are: upload & execute, query execution status and download. For the purpose of description, upload and execute can be seen as two different operations, where the execution automatically follows after an upload and is never called directly.

Upload. Depending on the realization of the shared directory, the upload may not be possible while the guest system is turned on. When the user uploads a file, the host deletes the current content of the shared directory, and then extracts the archive into this directory.

Execution. In order to provide the information for the status query, the script is not directly executed. Instead, a monitor program is called which then executes the script.

When uploading, there are three possible situations:

1. The guest system is turned off.
2. The guest system is turned on, and the host can invoke processes on the guest system.
3. The guest system is turned on, and the host cannot invoke processes on the guest system.

In case 1, the execution must be delayed until the guest system has been booted. On every guest system the monitor is executed at the boot process if the start script has been changed.

In case 2, the monitor is called by the host right after the archive has been extracted.

In case 3, the guest needs to run a daemon program which can react to changes in the shared directory. When a new start script appears, it executes the monitor. The same daemon may also handle case 1. In this case, the testbed must make sure that the daemon does not start the script before the archive has been completely extracted. One way of doing this is to not copy the start script into the shared directory before everything else is present.

Status Query. The status information consists of:

- Has the script finished? (*Done Flag*)
- Is the monitor still running? (*Running Info*)
- A custom string defined by the script (*Custom Status*)

This information is stored in a file called the status file, which is written by the monitor. The status file can be read by the host, which then provides the information to the testbed, which can make it accessible to the user.

The *Done Flag* will be set to true as soon as the monitor detects that the script process has terminated.

Since this termination cannot be detected if the monitor crashes or terminates before the script has been finished, the monitor repeatedly (i.e., every 2 minutes) writes the current timestamp into the *Running Info*. The host interprets this as a sequence number, and if it does not change for a certain amount of time, the monitor can be assumed to have stopped. Because the host only watches for changes, it is not necessary to synchronize the clocks. To hide complexity to the user, the host provides this information as a boolean value: The monitor is running or not.

The *Custom Status* can either be written by the start script, or the monitor provides a function which can be called by the script. This string may contain anything from a single value to an XML file. Since RexTFV provides an interface for the user (or any client program), this string can be used to send information from the virtual machine to the experimenter.

Additionally, the standard and error output of the script are being saved to the shared directory, where it can be downloaded as described in the next section.

Download. In order to download, the host packs the whole shared directory into an archive which can then be sent to the user.

This directory contains the start script's standard and error output, the status file, all the data which has been uploaded and not deleted, and all files which may have been generated in the shared directory by other programs and stored in this directory.

To avoid large downloads, the start script should delete unnecessary data like software packets after it has finished all other operations. In order to get all the necessary data, all programs should be configured to store their output data in this directory. If such a configuration is not possible, the start script must make sure to copy the data here after the experiment.

3.3 Architecture

RexTFV has been designed to not require any changes to the testbed's architecture, so that it can seemlessly integrated into an existing testbed by adding some function calls and adding these functions to the software controlling the hosts.

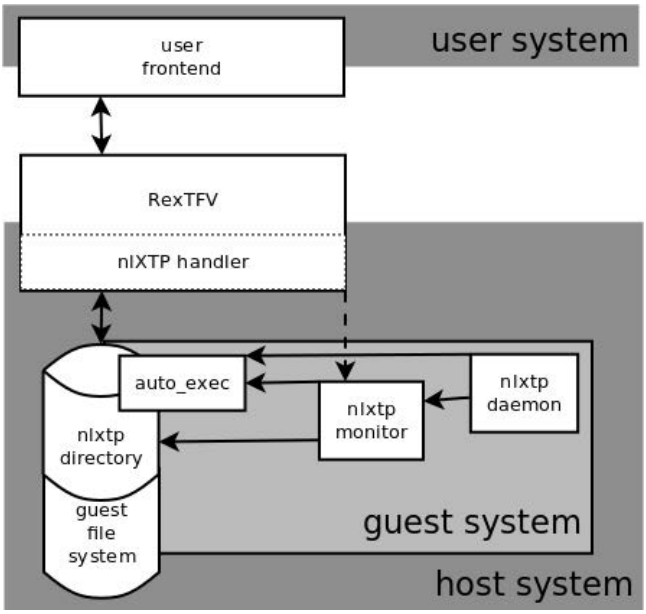

Fig. 3. Components of RexTFV using nlXTP and integration into the testbed

Figure 3 shows the distribution of components between guest, host and user system. Functions which are in the white area may be distributed as the testbed's architecture requires it. In general, the testbed must forward RexTFV function calls to the host system, and then use its nlXTP handler for communicating with the guest system, i.e. writing and reading from the shared directory, and eventually mounting and unmounting it. Since all function calls from the user to the nlXTP handler must run through the testbed, authentication and authorization for these operation can be checked by the testbed.

Function calls are always targeted at the host and never at the virtual devices. Thus, well-known technologies of network virtualization can be used to seperate this control-traffic from the traffic of the experiment in such a way that it becomes invisible for the experiment nodes. This way, this kind of control does not happen over the network from the point of view of the experiment nodes.

The operations from section 3.2 assume small programs on the guest system, the so-called "guest modules". These are the *nlXTP daemon*, which has to cover some cases for the auto-execution, and the *nlXTP monitor*, which has to execute the start script and write down the status information. In contrast to control over network, these requirements are low, because nlXTP does not require TCP/IP, SSH, user authentication or other complex programs on the devices, which are necessary for the core functionality.

If the guest modules are missing on a virtual machine, file transfers (the virtual floppy must be mounted manually), and the submission of status information (which must be written in the testbed-specific format in the status file)

are still possible in a manual way. This can also be used to install the guest modules manually on a newly created device. The only thing that would be impossible is the automatic execution of the start script.

4 Implementation

NlXTP has been implemented for the container-based OpenVZ and the full virtualization KVM. This proves that the concepts described in section 3 work. Since these concepts do not require or assume anything except the basic principles of container-based or full virtualization, they should work with other virtualization systems as well.

RexTFV has been implemented in ToMaTo using nlXTP. The functions can be found under the more user-friendly name *executable archives.*

5 Conclusion

RexTFV can be used to automate the lifecycle of devices in an experiment. When using nlXTP for host-to-guest and guest-to-host communication, it does not need any changes to the network configuration of a virtual machine, making it possible to run an experiment without ever connecting to the Internet, thus reducing noise from the outside which may affect the results. Furthermore, if an experimenter decides not to use RexTFV, its presence won't change the experiment's setup.

NlXTP makes use of the fact that the host and the guest system access the same physical storage to emulate a shared directory for network-less communication and is therefore only applicable in such a situation. It was specifically designed to avoid using an IP stack communication on experiment nodes.

Archives can not only be used for file transmission or single commands, but also for automating parts of or the whole experiment lifecycle on a testing node. In principle, the knowledge about which archives have been uploaded on which devices at what time in the experiment may, together with all testbed variables, determine the whole experiment. This can increase reproducibility and confirmability for the given experiment, if the archives are provided to the readers of a publication.

References

1. Albrecht, J., Huang, D.Y.: Managing Distributed Applications Using Gush. In: Magedanz, T., Gavras, A., Thanh, N.H., Chase, J.S. (eds.) TridentCom 2010. LNICST, vol. 46, pp. 401–411. Springer, Heidelberg (2011). http://link.springer.com/chapter/10.1007/978-3-642-17851-1_31
2. Bastin, N., Bavier, A., Blaine, J., Chen, J., Krishnan, N., Mambretti, J., McGeer, R., Ricci, R., Watts, N.: The instageni initiative: an architecture for distributed systems and advanced programmable networks. Computer Networks (2014). http://dl.acm.org/citation.cfm?id=2612045

3. Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., Bowman, M.: Planetlab: an overlay testbed for broad-coverage services. SIGCOMM Comput. Commun. Rev. 33(3), 3–12 (2003) ISSN 0146–4833. doi:10.1145/956993.956995. http://doi.acm.org/10.1145/956993.956995
4. Schwerdel, D., Hock, D., Günther, D., Reuther, B., Müller, P., Tran-Gia, P.: ToMaTo - A Network Experimentation Tool. In: Korakis, T., Li, H., Tran-Gia, P., Park, H.-S. (eds.) TridentCom 2011. LNICST, vol. 90, pp. 1–10. Springer, Heidelberg (2012). http://link.springer.com/chapter/10.1007/978-3-642-29273-6_1
5. Schwerdel, D., Reuther, B., Mueller, P.: Malware analysis in the tomato testbed (2011). http://dspace.icsy.de:12000/dspace/handle/123456789/350
6. Schwerdel, D., Reuther, B., Zinner, T., Mueller, P., Tran-Gia, P.: Future internet research and experimentation: The g-lab approach. Computer Networks 61(0), 102–117 (2014). ISSN 1389–1286. doi:http://dx.doi.org/10.1016/j.bjp.2013.12.023.. http://www.sciencedirect.com/science/article/pii/S1389128613004362 (Special issue on Future Internet) Testbeds - Part I
7. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. pp. 255–270, Boston, MA (December 2002). http://dl.acm.org/citation.cfm?id=844152