EAI Endorsed Transactions
on Industrial Networks and Intelligent Systems Research Article ICST.ORG

A networkless data exchange and control mechanism for virtual testbed devices

Tim Gerhard[1], Dennis Schwerdel[1], Paul Müller[1]

[1]Integrated Communication Systems Lab, University of Kaiserslautern, Germany

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

doi: 10.4108/inis.2.2.e2

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 [4] or EmuLab [2, 9]) use the devices' networking capabilities to provide such an interface, and 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, 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 [7]).

This paper proposes a new approach to control interfaces by not using of the testbed devices' network interfaces. Instead, another medium of communication will be chosen. 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 without using network interfaces between these two components, section 4 introduces the actual implementation in the Topology Management Tool (ToMaTo [6, 8]) 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, as will be described in section 3.2.

The *network-less Execution and Transfer Protocol* (nlXTP), which will be described in this section, 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 this will be needed in section 3.2, 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, which can be created and edited 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.

3.1.1. Container-Based Virtualization. Cantainer-based virtual machines (such as OpenVZ[1] or Linux-VServer[2]) 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.

3.1.2. Full Virtualization. In full virtualization systems (like KVM[3]/QEMU[4] [3, 5] or VirtualBox[5]), 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

[1] http://openvz.org
[2] http://linux-vserver.org
[3] http://linux-kvm.org
[4] http://qemu.org
[5] http://www.virtualbox.org

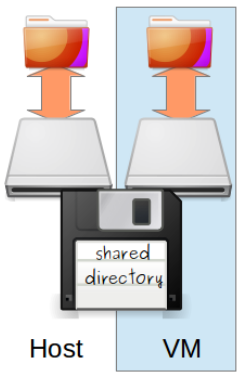

Figure 1. The virtual disk containing the shared directory can be mounted in both systems simultaneously.

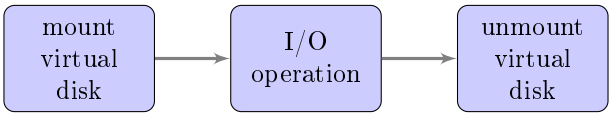

Figure 2. Access Sequence when the host performs an I/O operation on the virtual disk which may be mounted on a VM

down. This means that archive uploads are unavailable while the guest system is running.

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.

3.2.1. 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.

3.2.2. 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.

3.2.3. 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 the next 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.

3.2.4. 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.

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 from the user are always targeted at the host and never at the virtual devices. Thus, well-known technologies of network virtualization can be used

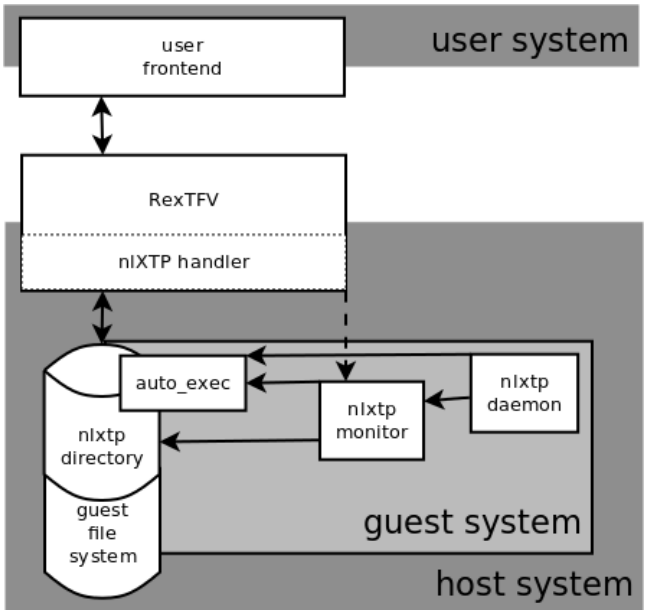

Figure 3. Components of RexTFV using nlXTP, and integration into a testbed's architecture. Function calls always start at the user frontend and are forwarded through the testbed to the nlXTP handler, which is a part of the testbed software on the host system. The guest system needs guest modules in order to provide the functionality.

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 to 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 without guest modules is the automatic execution of the start script.

4. Implementation and performance

4.1. 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[6] using nlXTP. The functions can be found under the more user-friendly name *executable archives*. ToMaTo has a central *backend* which controlls a number of *hosts* which are distributed over multiple sites worldwide. The backend is accessible via an open API by multiple *frontends*, most commonly the web frontend, which runs inside a browser. For uploading and downloading, the frontend acquires a *grant*, which can be used to do an upload or download via HTTP on a host. After uploading, the frontend can use an additional API command to call the function which extracts the archive to the shared directory, and, if needed, executes the monitor. Status information can be read from a VM's information, which is already accessible via the API. Grant-based HTTP access to hosts and VM information were already implemented, and the only changes which had to be made to the testbed were new control structures, the shared directory, extracting an archive to this shared directory, and creating an archive from this shared directory (including API calls and web frontend functions to access these). Also, device images had to be updated to include the nlXTP guest modules. ToMaTo is open source, and the implementation is available in the ToMaTo repository[7]

4.2. Performance

NlXTP and RexTFV do not have performance-relevant functions. The performance of uploads and downloads depend on the connection between the user and the host. The performance of the extraction and archive creation depends on how fast the used program or library can do this action on the host. Accessing the status information is as performant as accesing a file on the host's disk and the I/O overhead due to virtualization. RexTFV and nlXTP do not produce a significant overhead over these functions themselves, and all calls only involve at most one large network transmission, one compression or decompression, and one access to the status information. Querying the status information must be done in an interval in which it will not significantly interfere with other disk I/O operations.

[6] http://tomato-lab.org

[7] https://github.com/glab/ToMaTo - the most relevant files to this paper are: `hostmanager/tomato/elements/{__init__.py,kvmqm.py,openvz.py}` and `nlxtp_gest_modules/`

4.3. Experiments

The following three experiments have been conducted in the ToMaTo testbed in order to show the feasibility of and use cases for RexTFV. This means that it has been tested with OpenVZ as a container-based virtualization system, and KVM as a full virtualization system. All VMs used prepared system images on which the nlXTP guest modules had been installed. The default OS was Debian 7 Linux. Access to devices' consoles was possible via VNC, which is a built-in feature in the testbed. The testbed also offers an open API for node control.

In the first experiment, it was shown that the basic nlXTP functions work in the given implementation. The second experiment demonstrates that it is possible to run actual programs on the computer and use nlXTP/RexTFV to transmit files or install software. The last experiments demonstrates a two-way communication between a local computer, and a testbed device, over nlXTP.

4.3.1. Basic output functions. In the archive was a start script which first sleeps for a second, then sets a pre-defined character string as custom status, then writes another string into a file in the shared directory, and then writes another string into the standard output.

This archive has been uploaded to (1) a running OpenVZ device, (2) a stopped OpenVZ device, and (3) a stopped KVM device. After uploading, the stopped devices have been started. In all cases, the running indicator was true for about a second, then the done indicator was set to true and the custom status was set as expected. After this, a download of the archive has been executed. The downloaded archive contained, besides the script itself, the script's standard output in a file, the status file, and the file which has been written by the script. Standard output and this written file were the same as when the script was run outside the testbed.

After this, the VMs' consoles were accessed to check whether the shared directory exists on the devices, and their contents match the contents of the downloaded archive. This was confirmed in all three cases.

Since the script, and thus the archive, were small, and the script does not do much, everything happened without any noticable delay besides what would be expected without nlXTP (like booting the VMs). This confirms that auto-execution, upload, and download work as expected.

4.3.2. Installation of a Debian packet. The archive contained a small debian package file, containing zsh, and the start script installs this package using dpkg. The package file has been fetched from the Debian repositories. This experiment has been done on a running OpenVZ device.

Before uploading the archive, the device's console was accessed to confirm that the program was not already

installed on the device, and to install all dependencies. Then, the archive was uploaded. Uploading, extracting and running the script took a reasonable time with respect to the package size. After the script has finished executing, it was confirmed that the program was installed by running it on the device's console. This proves that it is possible to affect system configuration using RexTFV.

4.3.3. Automatic installer archive creation. In the third experiment, we wrote a script which creates installer archives for a program, including its dependencies, for multiple operating systems (OS). This script was running on a local computer and controlling the testbed. The basic algorithm was:

1. For each OS, create and start an OpenVZ device in the testbed, and connect this device to the Internet. Create a probing archive, and upload this to each device.
2. The probing archive's start script uses the custom status to show an OS identifyer, and a list of needed packet files including their download URL. This list also includedes dependencies which are not installed.
3. For each OS, fetch the custom status of the corresponding OpenVZ device, download all packet files to a packet directory, and make an installer list linking the OS identifyer to its corresponding packets.
4. Create an installer archive whose start script identifies the OS and installs the given packets. Include all packet files and installer lists from the previous step.

This has been tested with the packet openjdk-6-jre on Debian 6, Debian 7, Ubuntu 10.04 and Ubuntu 12.04, each for x86 and AMD64 processor architectures.

To get the needed information, the probing archive's start script first calculates the OS identifyer depending on distribution, version, and processor architecture. Then it detects the system's packet manager (in this test case, it could just test whether apt-get was available, which was true for all cases), and then used this packet manager to update the package database and then get the required package names and their URLs.

OS identifyer and packet manager detection were reused in the resulting installer archive.

The resulting installer archive included installer lists for all eight systems. It has been tested by uploading it to Debian 7 (both AMD64 and x86) and Ubuntu 12.04 (AMD64) devices. Success was tested in the same way as in the single installer experiment before. In case of Ubuntu, the packet was already installed; as expected, the installer list was empty, and the standard output of the script matched with this.

This experiment demonstrates communication between a local computer and (running) testbed devices. The query archive's start script invokes several commands to get OS identifier, packet manager, and packet lists, and there is theoretically nothing that prevents it to use any other program that is available on the machine. This means, that a local program could invoke any command on the testbed device.

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.

Since archives can be seen as functions with a certain purpose, they can be reused in other experiments. The testbed can also provide a set of archives which do frequently needed things, like installing a certain program. Such an archive was demonstrated in the last experiment.

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] Jeannie Albrecht and Danny Yuxing Huang. Managing distributed applications using gush. Proceedings of the Sixth International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities, Testbeds Practices Session (TridentCom), 5 2010.

[2] Nicholas Bastin, Andy Bavier, Jessica Blaine, Jim Chen, Narayan Krishnan, Joe Mambretti, Rick McGeer, Rob Ricci, and Nicki Watts. The instageni initiative: an architecture for distributed systems and advanced programmable networks. *Computer Networks*, 2014.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[4] 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, July 2003. ISSN 0146-4833. doi: 10.1145/956993.956995. URL http://doi.acm.org/10.1145/956993.956995.

[5] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[6] Dennis Schwerdel, David Hock, Daniel Günther, Bernd Reuther, Phuoc Tran-Gia, and Paul Müller. Tomato-a network experimentation tool. 7th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom 2011), 2011.

[7] Dennis Schwerdel, Bernd Reuther, and Paul Mueller. Malware analysis in the tomato testbed. 2011.

[8] Dennis Schwerdel, Bernd Reuther, Thomas Zinner, Paul Mueller, and Phouc Tran-Gia. 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. URL http://www.sciencedirect.com/science/article/pii/S1389128613004362
Special issue on FutureInternetTestbeds- PartI.

[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. pages 255–270, Boston, MA, December 2002.