electronics

MDPI

Article

Easy as Pi: A Network Coding Raspberry Pi Testbed

Chres W. Sørensen [1,*,†], Néstor J. Hernández Marcano [1,2,†], Juan A. Cabrera Guerrero [3,4], Simon Wunderlich [3], Daniel E. Lucani [1,†] and Frank H. P. Fitzek [3]

1 Department of Electronic Systems, Aalborg University, Aalborg 9220, Denmark; nestor@steinwurf.com (N.J.H.M.); del@es.aau.dk (D.E.L.)
2 Steinwurf ApS, Aalborg 9220, Denmark
3 Deutsche Telekom Chair of Communication Networks, Technische Universität Dresden, Dresden 01069, Germany; juan.cabrera@tu-dresden.de (J.A.C.G.); simon.wunderlich@mailbox.tu-dresden.de (S.W.); frank.fitzek@tu-dresden.de (F.H.P.F.)
4 SFB 912—Collaborative Research Center HAEC, Dresden 01069, Germany
* Correspondence: cws@es.aau.dk; Tel.: +45-99-408-723
† Current address: Fredrik Bajers Vej 7A, Room A3-110, Aalborg 9220, Denmark.

Academic Editors: Steven J. Johnston and Simon J. Cox
Received: 19 July 2016; Accepted: 28 September 2016; Published: 13 October 2016

Abstract: In the near future, upcoming communications and storage networks are expected to tolerate major difficulties produced by huge amounts of data being generated from the Internet of Things (IoT). For these types of networks, strategies and mechanisms based on network coding have appeared as an alternative to overcome these difficulties in a holistic manner, e.g., without sacrificing the benefit of a given network metric when improving another. There has been recurrent issues on: (i) making large-scale deployments akin to the Internet of Things; (ii) assessing and (iii) replicating the obtained results in preliminary studies. Therefore, finding testbeds that can deal with large-scale deployments and not lose historic data in order to evaluate these mechanisms are greatly needed and desirable from a research perspective. However, this can be hard to manage, not only due to the inherent costs of the hardware, but also due to maintenance challenges. In this paper, we present the required key steps to design, setup and maintain an inexpensive testbed using Raspberry Pi devices for communications and storage networks with network coding capabilities. This testbed can be utilized for any applications requiring results replicability.

Keywords: Linux; network coding; Raspberry Pi; testbed; C++

1. Introduction

Upcoming 5G technology is targeting the controlling and steering of the Internet of Things (IoT) in real-time on a global scale. This will break new ground for new markets such as driverless vehicles, manufacturing, humanoid robots, and smart grids. The number of wireless devices is expected to increase by five times to up to 50 billion devices [1]. It is generally believed that those devices will not be connected in the same manner as current devices are connected today. Centralized systems will collapse in terms of capacity, while distributed systems appear as an alternative. Therefore, we believe mesh technologies will play a major role in the communication architecture in future systems. Mesh technology has been known for sensor and ad hoc networks or mobile cloud scenarios, but the technical requirements on 5G mesh-based communication systems are dramatically increasing. Future mesh networks need to support high data rate, low latency, security, network availability and heterogeneous devices to ensure high Quality of Experience (QoE) for the final user. In state-of-the-art systems, those requirements are traded-off with each other, but in the 5G context, we cannot do this anymore.

Introduced by Ahlswede et al. [2], network coding constitutes a paradigm shift in the way that researchers and industry understand and operate networks, by changing the role of intermediate relays in the process of transmission of information. Relays are no longer limited to storing and forwarding data, but also take part in the coding process, through a process called recoding, where the relay generates new linear combinations of incoming coded packets without previously decoding the data. Network coding allows the increase of throughput, reliability, security and delay performance of the networks. In previous works, we have shown that Random Linear Network Coding (RLNC) [3,4] is able to satisfy the aforementioned technical requirements. We have actually shown how to increase the throughput [5], reduce the delay [6] or support heterogeneity for coding enabled communication nodes [7].

In our prior works, the C++11 Kodo library [8] was used as the common building block containing the basic RLNC functionalities. Most of the work was focusing on small mesh networks with a handful of communication nodes, though the expected scenarios are fairly beyond this order of magnitude. Despite this successful deployment in real systems, many of these protocols and contributions have been implemented in separate testbeds and the experiences are hard to reproduce. Deploying a large-scale and configurable testbed for networking and storage can be challenging, not only due to the inherent costs of the hardware, but due to maintenance challenges and ability to replicate results consistently. The latter requires not only the devices to run the same Operating System (OS), but also have exactly the same configurations and software packages. There is a need to evaluate large-scale network deployments of low-cost devices in a quick, easy-to-deploy, reproducible and maintainable fashion.

The emergence of powerful and inexpensive single-board computers opens new possibilities in this area. By running a standard OS, they allow implementations that are compatible with higher end devices. In addition, they utilize stable software supported by their communities. For example, the Iridis-pi platform [9] provides a detailed description of a Raspberry Pi (Raspi) [10] testbed ideal for educational applications. Here, the authors present computational speed benchmarks, inter-node communication throughput and memory card writing speeds for data storage to assess the testbed performance. This work indicates only a basic description of how to set up the required software and also mentions that its maintenance could be time-consuming. Moreover, this work does not consider possible network coding applications. Different studies of IoT applications consider using the Raspi for data processing: In [11], the Raspi is the processing unit that coordinates and controls the activity of an isle of lamps on a public road and reports it to a monitoring center. A use case regarding remote environment surveillance using the Raspi and the Arduino [12] technologies is presented in [13]. Here, both devices report air pressure, humidity and temperature of the locations of cultural paintings plus high-resolution images of the paintings themselves. This data is sent to a monitoring center to ensure the preservation of the paintings. Furthermore, authors in [14] consider FingerScanner, a technology that utilizes the Raspi to act as the data server in a finger scanning application that collects the fingerprints. Even though all these applications consider the use of the Raspi as a core block, they provide few to no descriptions of their procedures to configure the Raspi. These applications become cumbersome to maintain as their considered systems could potentially scale when aiming to serve more users. The current way that the data is sent in the considered networks for these IoT applications will not be feasible in future 5G systems as mentioned previously.

Given this set of specific needs, in this work, we present the design, key step-by-step instructions and mechanisms to setup, configure and maintain an inexpensive testbed using potentially several Raspi devices for networking (wireless or wired) and storage applications including RLNC functionalities into the testbed through Kodo. The architecture itself is not bounded to the networking area and can be used for other applications that require replicable results with the Raspi. Our work for the testbed procedure is organized as follows: Section 2 introduces the testbed system. In Section 3, we provide details about the testbed setup, scripts, configuration files and connectivity. In Section 4, we elaborate on the need and setup for an overlay filesystem for our testbed in order to have both persistent and non-persistent data on it as an optional step. Section 5 describes a set of automation and monitoring tools that can be included in the testbed to simplify the execution of routinary and repetitive tasks. Section 6 elaborates on the compilation of the Kodo library for the Raspi. Conclusions and future work are reviewed in Section 7. Finally, a set of alternative commands, in case the ones presented in this work might not be executed, are discussed in the Appendices.

2. Testbed Overview and Design Criteria

A sketch of the testbed is depicted in Figure 1. The testbed consists of up to 100 Raspis of different models. More specifically, in our design, we consider: Raspberry Pi 1 model B rev. 2, Raspberry Pi 2 model B V1.1 and Raspberry Pi 3 model B V1.2. All Raspis are each equipped with a 8 GB Secure Digital (SD) memory card, a wired and wireless network interface and a power supply. All the Raspi are connected to a common Local Area Network (LAN) that provides internal and external connectivity. Without loss of generality, in our case, they are connected to a university network using their wired Ethernet interface that is named `eth0` according to the legacy naming convention of Ethernet interfaces in Linux [15]. We consider the university network since our testbed is used by students and academic staff to perform measurements and experimentation of controlled and reproducible scenarios as part of academic research. The testbed description and procedures for setting it up are not restricted to this academic scenario. All Raspis are configured to run a Secure Shell (SSH) daemon for easy remote access within the university network. We requested the university Information Technology (IT) department to configure the university Dynamic Host Configuration Protocol (DHCP) server to assign each Raspi a static Internet Protocol (IP) address. This eliminates the demand for monitors and keyboards with the Raspis for non-graphical applications. Finally, our design aims to configure all Raspis identically from a customized bootable image in their respective memory cards, while still allowing the end-users to store files locally in each of the Raspis.

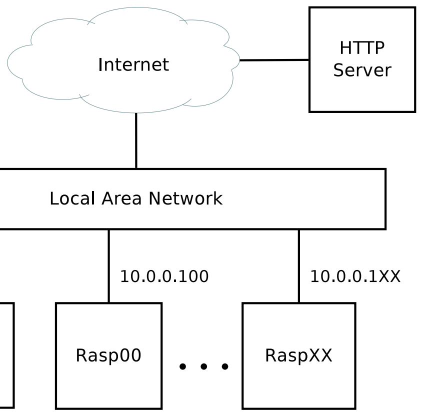

Figure 1. Testbed setup.

We will refer to the testbed administrator as the person(s) in charge of setting up and configuring the testbed with administrator privileges from the OS point of view. The setting and configuration procedures are performed by the testbed administrator in a PC running a Linux distribution as shown in Figure 1. Although in principle the administrator Linux distribution is not a restriction, we present our procedure in a Debian-based Linux distribution. Our basic design considers to create a customized image to store it later on a memory card for each Raspi. Once configured, we store the resulting image file in a Hyper Text Transfer Protocol (HTTP) server as backup and in case the testbed administrator requires the making of new changes to this file. In our case, we store all files at Zenodo [16], but the testbed administrator should copy the our files to his/her own HTTP server to get read/write permissions. We also put all the required configuration files and scripts for the Raspis setup in the HTTP server so there is a single place where system setup is stored and could be modified. This simplifies the system maintenance, as it may not always be desirable to make persistent changes on the Raspis—for example, when different users are interested in running experiments on a rebooted testbed. We later present how to utilize stacked filesystems to enable both persistent and temporary storage to have this capability. Its purpose is to remove non-desired data after a reboot while keeping the original customized image structure. This step of the procedure is optional if the testbed administrator decides to keep only persistent changes regardless of the testbed use. Finally, we include a set of automation, monitoring and cross-compilation tools over the top of our system in order to simplify the execution of repetitive and long tasks, be able to follow the progress of long task processes and compile relevant C++ source code for the testbed administrator.

3. OS Image Setup

In this section, we review the steps to create a common OS image for all the Raspis. The image setup is composed of three major steps: select and download the OS image file, alter the image structure and configure the OS files. We proceed to detail all these steps providing brief discussions to our setup choices when required. To perform these steps, we indicate with command-line blocks the required sequential commands to be typed by the testbed administrator on his/her PC to obtain the desired setting. In all the command blocks in the paper, we indicate if a command needs to be run with root permissions (#) or common user permissions ($). These signs will prefix the commands.

3.1. OS Selection and Download

To get started, we first need to install an OS that works properly on all the Raspi models. We will download and setup the image in the testbed administrator PC using a Debian-based distribution. An alternative to this method, is to create a tailoared Linux distribution for the Raspi platform using the Yocto Project [17]. However, this process would require assembly and compilation of all the software for the Raspi platform from scratch, which goes beyond the scope of our work. We use the popular Debian-based Raspbian Linux [18] given that is the recommended and default OS for the Raspi. Raspbian is made available in two bundles: Raspbian and Raspbian Lite. The difference between the two is that Raspbian contains a pre-installed desktop environment for user interaction, and Raspbian Lite by default only permits interaction through a command shell. Given that the Raspis in our testbed are not connected to monitors, we decide to work with Raspbian Lite. If required, a desktop environment can be installed using the package manager later.

The latest Raspbian Lite bundle can be downloaded from the Raspbian official webpage [18]. At the time of this writing, the latest available bundle was `2016-05-27-raspbian-jessie-lite.zip`. To ensure that the content of the bundle does not change, this procedure is based on that particular version of Raspbian Lite, which we have made available at [16]. All other files used in this paper are also available there. The testbed administrator has to move these files to his/her own HTTP server. To get started, the testbed administrator must open a Linux shell (terminal) on his/her PC and declare the environment variables shown in the command block below. We show the whole procedure by performing the role of the testbed administrator.

```
$ export URL="https://zenodo.org/record/154328/files/"
$ export IMAGE="2016-05-27-raspbian-jessie-lite"
$ export WORKDIR="${HOME}/Raspbian"
```

In this code block, the `${URL}` and `${IMAGE}` variables specify where the Linux bundle is located and `${WORKDIR}` specifies a working directory where the Raspbian Lite bundle will be downloaded and customized. If the testbed administrator allocates his/her files into another location, then it will be required to change the `${URL}` environment variable. Notice that even though we use the $ and # signs in the shell, in general, these signs will be particular to the testbed administrator OS shell. Next, we create the working directory and change to it with the `cd` command. To download the image, we utilize the `wget` command before unpacking the `zip` file as follows:

```
$ mkdir -p ${WORKDIR}
$ cd ${WORKDIR}
$ wget ${URL%/}/${IMAGE}.zip
$ unzip ${IMAGE}.zip
```

3.2. Image Customization

After Raspbian Lite has been unpacked, there should be an `.img` file in the working directory `${WORKDIR}`. `fdisk` can be used to display the content of the image. We parse the arguments `-u sectors` to display the sizes in sectors and `-l` to display the partitions within the image. The `fdisk` command should output to the terminal something similar to:

```
$ fdisk -u=sectors -l ${IMAGE}.img
Disk 2016-05-27-raspbian-jessie-lite.img: 1.3 GiB, 1387266048 bytes, 2709504 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x6fcf21f3

Device                               Boot  Start     End Sectors  Size Id Type
2016-05-27-raspbian-jessie-lite.img1        8192  137215  129024   63M  c W95 FAT32 (LBA)
2016-05-27-raspbian-jessie-lite.img2      137216 2709503 2572288  1.2G 83 Linux
```

The output provides relevant information about the image. The image is in total 2,709,504 sectors (1.3 GiB) in size and contains two partitions. The first partition starts at sector 8192 and the other partition starts at sector 137,216. The first partition type is FAT32 with a size of 63 MB and the second partition is of type Linux with a size of 1.2 GB. This indicates that the first partition is a boot partition, and the second one is a traditional Linux filesystem. In this case, the root filesystem, i.e., /.

3.3. Image Resizing

Given that we want to customize the root filesystem in the Raspis, we need to expand the image file since 1.2 GB might not be enough to store the existing root filesystem plus additional files and software packages. Thus, we need to increase the partition size. The following procedure illustrates how the image and its root filesystem can be expanded by one GB. First, to expand the image one GB, we execute:

```
$ dd if=/dev/zero bs=1M count=1024 >> ${IMAGE}.img && sync
```

Later, we use `fdisk` with the same arguments as before to see that the image is now one GB larger:

```
$ fdisk -u=sectors -l ${IMAGE}.img
Disk 2016-05-27-raspbian-jessie-lite.img: 2.3 GiB, 2461007872 bytes, 4806656 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x6fcf21f3

Device                               Boot  Start     End Sectors  Size Id Type
2016-05-27-raspbian-jessie-lite.img1        8192  137215  129024   63M  c W95 FAT32 (LBA)
2016-05-27-raspbian-jessie-lite.img2      137216 2709503 2572288  1.2G 83 Linux
```

Now, in the above command block output, we observe that the change has taken effect by noticing the total available image size is 2.3 GiB. To expand the root filesystem, we replace the Linux partition with a new partition one GB larger. The starting point of this new partition should be the same as the old one. We make use of `fdisk` to alter the partition table in the commands below. They (i) delete partition number 2; (ii) create a new primary partition and (iii) set the new partition starting point. The new partition starting point value is 137,216 in our case; Finally, we (iv) write the new partition table to the image file. This is made as follows:

```
$ fdisk ${IMAGE}.img << EOF
d
2
n
p
2
137216

w
EOF
```

If the partitions commands were correct, the partition table should now look like the following:

```
$ fdisk -u=sectors -l ${IMAGE}.img
Disk 2016-05-27-raspbian-jessie-lite.img: 2.3 GiB, 2461007872 bytes, 4806656 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x6fcf21f3

Device                              Boot  Start     End Sectors  Size Id Type
2016-05-27-raspbian-jessie-lite.img1       8192  137215  129024   63M  c W95 FAT32 (LBA)
2016-05-27-raspbian-jessie-lite.img2     137216 4806655 4669440  2.2G 83 Linux
```

3.4. Loopback Device Setup

After successfully resizing the image file, we use a loopback device to make the Raspbian image available as a block device in the filesystem. For this command to work, the testbed administrator distribution must have the `util-linux` package with version 2.21 or higher. Otherwise, the `-P` argument of `losetup` will appear as invalid. If the version of `losetup` can not be updated for some reason, an alternative option for this part is presented in Appendix A.1 of the Appendices.

```
$ export DEV=$(sudo losetup --show -f -P ${IMAGE}.img); echo $DEV
/dev/loop0
```

If the previous command was succesful, the `lsblk` command can be used to list the available block devices in the filesystem as follows:

```
# lsblk
NAME      MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
...
loop0       7:0    0  2.3G  0 loop
|-loop0p1 259:2    0   63M  0 loop
|-loop0p2 259:3    0  2.2G  0 loop
...
```

The image block device appears as `/dev/loop0`. This block device has two partitions associated with it, e.g., `loop0p1` and `loop0p2`. Finally, we check the filesystem of the block device with `e2fsck` and resize it with the `resize2fs` command:

```
# e2fsck -f ${DEV}p2
e2fsck 1.42.8 (20-Jun-2013)
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
/dev/loop0p2: 35392/80480 files (0.1% non-contiguous), 201968/321536 blocks
# resize2fs ${DEV}p2
resize2fs 1.42.8 (20-Jun-2013)
Resizing the filesystem on /dev/loop0 to 583680 (4k) blocks.
The filesystem on /dev/loop0 is now 583680 blocks long.
```

3.5. Block Device Mounting

For browsing and altering the files in the image, we mount the block device partitions into a particular path of our `${WORKDIR}` in order to customize them. We mount the block device partition that contains the root filesystem and later the boot partition. This is done by creating an empty directory that is used as a mountpoint. We name it `root` and create it in the working directory before mounting the root filesystem onto the mountpoint. We mount the root filesystem as follows:

```
$ export ROOTDIR="${WORKDIR}/root"
$ mkdir -p ${ROOTDIR}
# mount ${DEV}p2 ${ROOTDIR}
```

The root filesystem mounted in `${ROOTDIR}` already has a boot directory that can be used as the mount point for the boot partition in the block device `/dev/loop0p1`. This is convenient because the final edited partition from `${ROOTDIR}/boot` will be mounted on this same directory when a Raspi starts up with a memory card containing the raw final image. Hence, to mount boot partition we do:

```
# mount ${DEV}p1 ${ROOTDIR}/boot
```

In this way, it is now possible to change all files within the Raspbian image as desired by editing the files in `${ROOTDIR}`. We take advantage of this to edit configuration files, append new files and even update and install packages.

3.6. Image OS Files and Configuration Scripts Setup

In general, the Raspis should be setup as similarly as possible. However, some particularities exist to differentiate the devices in principle. In addition, scripts containing further configurations for the Raspis are desirable to be distributed as part of the common image. Therefore, we present here the steps to setup basic properties of the Raspis and distributing configuration scripts to each of them through the image. For this, we first indicate how to obtain and put our configuration scripts in the image. Later, we describe the tasks performed by these configuration scripts. Finally, we indicate how and in which order the scripts are executed to configure all the devices. Any testbed administrator might modify or include other tasks according to his / her needs as we will show.

3.6.1. Image Default Configuration Scripts Download

In our case, we have our default configuration scripts stored in a file `rasp_config.zip` located in the same URL of the HTTP server where the image was retrieved from, i.e., the one in the environment variable `${URL}`. We first download this compressed file with `wget` and extract it locally into our Raspbian Lite image. These commands and the output of the last one are shown as follows:

```
$ wget ${URL%/}/rasp_config.zip
$ unzip rasp_config.zip -d ${ROOTDIR}/home/pi/
Archive:  rasp_config.zip
   creating: ${ROOTDIR}/home/pi/rasp_config/
  inflating: ${ROOTDIR}/home/pi/rasp_config/nodes.csv
  inflating: ${ROOTDIR}/home/pi/rasp_config/set_hostname
  inflating: ${ROOTDIR}/home/pi/rasp_config/main
  inflating: ${ROOTDIR}/home/pi/rasp_config/update_rasp_config
```

The unzippped files are one configuration file and three configuration scripts in the newly created `${ROOTDIR}/home/pi/rasp_config/` folder in the image. We describe which features that we require all the Raspis to have and how are they achieved with these configuration scripts.

3.6.2. Device Hostnames

The hostname helps the user to physically distinguish the devices from each other. In our case, we require the devices in our testbed to have different hostnames. We define the hostnames based on the Medium Access Control (MAC) addresses of the Raspis wired Ethernet interface.

Prior to this stage, the MAC address of a network card can be found using the command `ifconfig` or `ip addr` on a given Raspi. We store the MAC addresses and hostnames of the Raspis in the configuration file `${ROOTDIR}/home/pi/rasp_config/nodes.csv`. A sample of our file is shown as follows:

```
${ROOTDIR}/home/pi/rasp_config/nodes.csv
----------------------------------------
# Ethernet MAC    Hostname
b8:27:eb:5b:da:20 rasp00
b8:27:eb:7b:c3:91 rasp01
b8:27:eb:54:9c:64 rasp02
b8:27:eb:95:bd:11 rasp03
...
```

The testbed administrator has to insert the MAC addresses and hostnames of his/her Raspis obtained previously in the format shown in the configuration file. For each given Raspi, there is a MAC address and the corresponding hostname. This file will be employed by the `${ROOTDIR}/home/pi/rasp_config/set_hostname` Bourne Again SHell (Bash) script to assign the hostname of each Raspi. The script content is the following:

```
${ROOTDIR}/home/pi/rasp_config/set_hostname
-------------------------------------------
#!/usr/bin/env bash

script_path="$(dirname $(realpath $0))"
config_file=${script_path}/nodes.csv
mac=$(cat /sys/class/net/eth0/address)
old_hostname=$(hostname)
new_hostname=$(grep $mac $config_file | cut -f2 -d' ')

# Assign hostname found in nodes.csv
if [ ! -z ${new_hostname} ]; then
    echo ${new_hostname} > /etc/hostname
    hostname ${new_hostname}
    sed -i.old -e "s:${old_hostname}:${new_hostname}:g" /etc/hosts
fi
```

The script (in lines): (1) tells the system to interpret the script using Bash; (3–4) gets the path to the script itself and the list of hostnames; (5) gets the MAC address of the node itself; (6) gets the current hostname; (7) gets the new hostname from the hostname list; and (10–14) assigns the new hostname to the Raspi where the script will be executed.

3.6.3. Updating Default Configuration Files and Scripts

Besides the single script with its configuration file introduced up to this point of our procedure, it is possible that the testbed administrator may require to add other scripts to configure his/her Raspis. We want to ensure that all the Raspi configuration scripts of any testbed administrators are obtained in a simple way. We automate this task by including the `${ROOTDIR}/home/pi/rasp_config/update_rasp_config` script in our procedure. The purpose of this script is to make all the Raspis fetch all the configuration scripts located with the image during a testbed start up.

In our case, as the testbed administrator for presenting the procedure, we want to fetch all our configuration scripts in `${URL%/}/raspi_config.zip`. The update script *automatically* downloads all the required configuration files in `rasp_config.zip` file from a remote location. This is the same that we manually did earlier to get our files, but this will be made in an automated way after booting up the system. This script content is:

```
${ROOTDIR}/home/pi/rasp_config/update_rasp_config
-------------------------------------------------
1  #!/usr/bin/env bash
2
3  url="https://zenodo.org/record/154328/files/"
4  config_file="rasp_config.zip"
5
6  # Attempt to fetch new configuration files
7  if ! wget -q --show-progress -O /tmp/${config_file} ${url%/}/${config_file}; then
8      echo "Warning: Unable to update rasp_config files"
9      exit 1
10 fi
11
12 # Unzip and overwrite configurationn files to root's home directory
13 unzip -q -o /tmp/${config_file} -d /home/pi/
```

The update script lines (3–4) specify the URL and `.zip` file that should be downloaded. Lines (7–10) download the configuration files to `/tmp` folder in the corresponding Raspi. It also prints a warning in case of errors and line (13) unzips the files to `/home/pi/`, the Raspi home directory. Existing files and directories are simply overwritten.

For the above scripts to work in the Raspis, it is required that the Raspis MAC addresses are found in `nodes.csv`. In addition, it should be noted that for other testbed administrators besides ourselves, the URL for file fetching and the configuration scripts themselves can be modified to fit their requirements. If required for a testbed administrator, the `rasp_config.zip` will need to be edited to include all the required configuration files and scripts. In addition, it might be necessary to edit the URL in the script `update_rasp_config` to store and fetch from a different location. Nevertheless, both the URL and configuration files presented here can be used as a starting boilerplate if desired.

3.6.4. Configuration Scripts Execution Order

To actually make the Raspis change hostnames and any other considered configurations, we have to make each Raspi call the above scripts when it starts up. After finishing the setup process, all the unzipped files presented in Section 3.6.1 should be locally available at each Raspi after getting the root filesystem. We first need to run the update script before running any other configuration scripts. To do this after boot up, we include a call for the update script in `${ROOTDIR}/etc/rc.local` before `exit 0` in the file:

```
# sed -i '/^exit 0/i bash /home/pi/rasp_config/update_rasp_config' ${ROOTDIR}/etc/rc.local
```

If it is required to have more configuration scripts, adding them in the `rc.local` file makes maintenance by the testbed administrator difficult since this needs to be both in the image and the downloaded `rasp_config.zip`. To avoid this problem, we include the `${ROOTDIR}/home/pi/rasp_config/main` script that calls all other configuration scripts (besides `update_rasp_config`) in a sequential order. This script content is:

```
${ROOTDIR}/home/pi/rasp_config/main
-----------------------------------
#!/usr/bin/env bash

bash /home/pi/rasp_config/set_hostname
# Any other required configuration scripts...
```

In this way, the automation process is simplified since we do not need to modify `${ROOTDIR}/etc/rc.local` again after the image has been written to the memory cards. Now, we insert a call to the `main` script in `${ROOTDIR}/etc/rc.local` as follows:

```
# sed -i '/^exit 0/i bash /home/pi/rasp_config/main' ${ROOTDIR}/etc/rc.local
```

Finally, `${ROOTDIR}/etc/rc.local` should look like the following:

```
${ROOTDIR}/etc/rc.local
-----------------------
...
bash /home/pi/rasp_config/update_rasp_config
bash /home/pi/rasp_config/main
exit 0
```

Notice that `set_hostname` is now called by the `main` script instead. The update script is still called directly. This ensures that all configuration scripts are updated before executed. Changes to the update script itself will first take effect at the next system startup.

3.7. Image Package Updating by Changing the Apparent Root Directory

Besides adding and configuring files within the image, the testbed administrator may want to install and update the software packages within the image before it is written to all the memory cards that goes into the Raspis. From any Linux x86 machine as the testbed administrator PC, this can be done using `chroot` command in the Quick Emulator (QEMU) [19] hypervisor for Advanced RISC Machine (ARM) processors.

`chroot` is a method in Linux that modifies the apparent root filesystem location from / to any other path. Consequently, in our case, we can use the Raspbian Lite image root filesystem within the testbed administrator Linux distribution. Then, QEMU allows the execution of commands for the Raspi image (ARM instructions) through the ones from the testbed administrator PC architecture. Due to the ARM processor that the Raspis employ, installation of the QEMU related software is required first and verification that QEMU is ARM enabled. To do so, run the following commands:

```
# apt-get install binfmt-support qemu qemu-user-static
# update-binfmts --display qemu-arm
qemu-arm (enabled):
     package = qemu-user-static
        type = magic
      offset = 0
       magic = \x7fELF\x01\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x28\x00
        mask = \xff\xff\xff\xff\xff\xff\xff\x00\xff\xff\xff\xff\xff\xff\xff\xff\xfe\xff\xff\xff
 interpreter = /usr/bin/qemu-arm-static
    detector =
```

In the previous output, the testbed administrator must be sure that the second command writes `qemu-arm (enabled)` as indicated. If that is not the case, then it should be possible to enable it by running:

```
# update-binfmts --enable qemu-arm
```

Provided that `qemu-arm` is enabled, we should now be able to `chroot` into our Raspbian lite image. There are a few commands to be performed before actually changing root into the root partion of the image. First, to get internet access from within the Raspbian lite image, copying the testbed administrator Linux distribution `resolv.conf` file into the image root filesystem is required. To do this, it is necessary to run the following:

```
$ cd $ROOTDIR
# cp /etc/resolv.conf ${ROOTDIR}/etc/resolv.conf
```

Now, because of the ARM architecture, the `/usr/bin/qemu-arm-static` command needs to be copied into the image before continuing by running:

```
# cp /usr/bin/qemu-arm-static ${ROOTDIR}/usr/bin
```

Before changing the root, it is necessary to populate the directories `proc`, `sys` and `dev` for the image to get control as the testbed administrator apparent root filesystem. This is made by the following commands:

```
# mount  -t proc proc proc/
# mount --bind /sys sys/
# mount --bind /dev dev/
# mount --bind /dev/pts dev/pts
```

Finally, run the following command to change root:

```
# chroot ${ROOTDIR} /usr/bin/qemu-arm-static /bin/bash
```

If successfully executed, our terminal should have changed the prompt, indicating that we are the root user in the Raspbian lite root filesystem as the apparent root. In case the `chroot` command is not successful, we provide an alternative command in Appendix A.2 of the Appendices. To be aware of the mode that we are working now, we change the prompt title to indicate that it is a `chroot` environment as follows:

```
# export PS1="(chroot) $PS1"
```

The Raspbian lite image should now be possible to use almost as if it had been booted in a Raspi. A major difference is that the testbed administrator PC is likely significantly faster than a Raspi. Hence, enabling updates, upgrades and installing new software packages should be faster than in a Raspi. Still updating and upgrading the packages for the Raspi might take some amount of time. To update the system package list, run the following command:

```
(chroot) # apt-get update
```

We further install some packages that we consider useful:

```
(chroot) # apt-get install vim git screen
```

`vim` is the improved `vi` editor for Linux, `git` for managing Git repositories and `screen` [20] for better handling of long-runnning processes. When writing the image to a memory card, all the changes that have been made to the image so far will exist in all Raspis after fetching it.

4. Overlay File System

In principle, our procedure modifies the image file only once in the testbed administrator PC when its setup is made. In addition, keeping this image in the Raspis provides the same initial system for all the devices. If we do not make any further modifications during the image setup, any files created after the initial boot of a Raspi will remain in the memory card. This is cumbersome to maintain since the size of the memory card is relatively small (8 GB), and there might be various users utilizing the testbed. In addition, different testbed users could be interested in running their experiments in a fresh rebooted system with the original customized image. We emphasize that this step is not necessary if the tesbed administrator wants to consider only persistent storage for its devices. A use case for this scenario could be a single user for the testbed or when a testbed administrator only wants to setup a few Raspis.

If both persistent and non-persistent storage are required for the Raspis, we present here the steps to setup an overlay filesystem. This type of filesystem enables an *upper* filesystem to overlay into a *lower* filesystem. Whenever a file is requested, the upper filesystem will forward the request to the lower filesystem in case it does not have it itself. If the upper filesystem has the requested file, it will simply return the file. This idea can be used in our setup to mount the root filesystem (i.e., Raspbian Lite) in the Raspis during startup as read-only filesystem. On the one hand, the image configuration files will remain after a reboot but the local data in these directories will be erased after a reboot. To enable the possibility of persistent changes, we overlay the upper filesystem that is mounted in the Raspi Random Access Memory (RAM), i.e., `/tmp` as rewritable on top of the lower root filesystem. Reading a file may return a file from the lower filesytem, but if it is stored, it will be saved in the upper filesystem. Accessing this file again will return the stored file from the upper layer. After a reboot, all the stored files in the upper filesystem will be retrieved, but the ones in the lower filesystem that are not part of the original image will be removed.

4.1. Filesystem Installation

Assuming that we are still in the `chroot` environment of the Raspbian Lite root filesystem for installing packages, we can setup the overlay filesystem at this point of the procedure. There already exists implementations overlaying the root filesystem. We use an implementation available at the Git repository in [21]. Since we have installed `git` in a previous step, we clone the repository. The command block below stores it in `/tmp` which is really mounted in RAM. All the files stored here will disappear when the system is rebooted.

```
(chroot) # OVERLAYROOTDIR="/tmp/overlayroot"
(chroot) # git clone https://github.com/chesty/overlayroot.git ${OVERLAYROOTDIR}
```

Before enabling the overlaying filesystem, it is necessary to generate an initial RAM filesystem or `initramfs`. This is an initial filesystem that is loaded into RAM during the startup process of a Linux machine to prepare the real filesystem. For this purpose, we need the BusyBox package by running:

```
(chroot) # apt-get install busybox
```

To create and activate the overlaying filesystem, we need to first add the required system scripts to do so. This is done as follows:

```
(chroot) # cp ${OVERLAYROOTDIR}/hooks-overlay /etc/initramfs-tools/hooks/
(chroot) # cp ${OVERLAYROOTDIR}/init-bottom-overlay /etc/initramfs-tools/scripts/init-bottom/
(chroot) # echo "overlay" > /etc/initramfs-tools/modules
```

To generate the initial RAM filesystem, we have to utilize the `mkinitramfs` command. This searches by default for the available kernel modules in the system. Since we are in `chroot` mode, we need to specify the correct kernel modules to search for. The available kernel modules are located in `/lib/modules`. To see them, we just run:

```
(chroot) # ls /lib/modules/
4.4.13+  4.4.13-v7+
```

Now, the initial RAM filesystem can be generated. Raspi version 1 needs a different kernel than Raspi version 2 and version 3. Kernel version 4.4.13+ is for Raspi version 1 and kernel 4.4.13-v7+ for Raspi version 2 and version 3. We proceed to generate an initial RAM filesystem for these kernels by running:

```
(chroot) # mkinitramfs -o /boot/init.gz -k 4.4.13+
(chroot) # mkinitramfs -o /boot/init-v7.gz -k 4.4.13-v7+
```

Although these commands might output some warnings, they should successfully generate working initial RAM filesystems. Later, an initial RAM filesystem will need to be called by the bootloader. In Raspbian, this is done by adding a command to `config.txt` file in the boot partition. If the system should be run in a Raspi version 1, then use `init.gz` by executing only the first code line below; otherwise, use `init-v7.gz` by executing only the second code line:

```
(chroot) # echo "initramfs init.gz" >> /boot/config.txt     # For Raspberry Pi version 1
(chroot) # echo "initramfs init-v7.gz" >> /boot/config.txt  # For Raspberry Pi version 2 or 3
```

After this point, it is no longer required to be in `chroot` mode. The following commands exit the `chroot` environment, unmount all partitions and detach the loopback devices:

```
(chroot) # exit
# cd ..
# umount --recursive ${ROOTDIR}
# losetup -d ${DEV}
```

For the `--recursive` option to work properly, it is necessary that the package `util-linux` version is greater than or equal to 2.22. Otherwise, an alternative is to either update the package or follow the procedure in Appendix A.3 of the Appendices.

4.2. Persistent and Non-Persistent Image Directories

Provided the stacked filesystem is configured, it is now possible to have directories where files are removed or not upon rebooting the Raspis. The following procedure creates an extra partition in the image for the Raspi user home directory that will be made storage persistent. We first expand image according to the desired home directory size, but avoid to making the image bigger than the target memory card size.

```
$ dd if=/dev/zero bs=1M count=1024 >> ${IMAGE}.img && sync
```

We create a partition for the home directory after the root partition. To do this, we again use `fdisk` to find the next available sector in the image. To verify the new available space for the full image and observe the next available sector, we run:

```
$ fdisk -u=sectors -l ${IMAGE}.img
Disk 2016-05-27-raspbian-jessie-lite.img: 3.3 GiB, 3534749696 bytes, 6903808 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x6fcf21f3

Device                               Boot  Start     End Sectors  Size Id Type
2016-05-27-raspbian-jessie-lite.img1        8192  137215  129024   63M  c W95 FAT32 (LBA)
2016-05-27-raspbian-jessie-lite.img2      137216 4806655 4669440  2.2G 83 Linux
```

We notice that one GB is now available to be used in the partitions. In addition, we observe the new partition should start at sector 4806656. To create it, we use `fdisk` as follows:

```
$ fdisk ${IMAGE}.img << EOF
n
p
3
4806656

w
EOF
```

We create a loopback device again and format the new partition, as follows:

```
# export DEV=$(sudo losetup --show -f -P ${IMAGE}.img); echo $DEV
/dev/loop0
# mkfs.ext4 ${DEV}p3
```

If the `-P` option is not available for `losetup`, we provide an alternative command line in Appendix A.1. Finally, if the previous filesystem formatting was successful, the filesystem is now available for use. We need to inform the Raspbian OS to mount the home partition that we have just created. This can be done by adding an entry in `fstab` as follows:

```
# mount ${DEV}p2 ${ROOTDIR}
# sed -i '$a /dev/mmcblk0p3 /home ext4 defaults,noatime 0 2' ${ROOTDIR}/etc/fstab
```

If the last command was executed correctly, the `${ROOTDIR}/etc/fstab` file should have the new line. The resulting file should look like the following:

```
${ROOTDIR}/etc/fstab
--------------------
proc              /proc            proc    defaults          0       0
/dev/mmcblk0p1    /boot            vfat    defaults          0       2
/dev/mmcblk0p2    /                ext4    defaults,noatime  0       1
/dev/mmcblk0p3    /home            ext4    defaults,noatime  0       2
```

Originally, the home folder is located in the root filesystem. However, we have to move its content to the new home partition and store it properly. We do that as follows:

```
# mount ${DEV}p3 ${ROOTDIR}/mnt
# mv ${ROOTDIR}/home/* ${ROOTDIR}/mnt/
```

Now, unmount again all the partitions and detach the loop devices as follows:

```
# umount --recursive ${ROOTDIR}
# losetup -d ${DEV}
```

If the `--recursive` option is not available, then follow the procedure in Appendix A.3 of the Appendices. If the steps are successfully executed up to this point, the customized image is available in the `${IMAGE}.img` file and is ready to be deployed into the Raspis. In the following section, we indicate how to proceed with the writing of the image into various memory cards.

4.3. Writing Customized Image to SD Memory Cards

For a basic system setup, the final step is to write the customized image to all the memory cards before they can be used in the Raspis. For our current considered system, we do this manually for each card. The testbed administrator needs to insert each memory card in his/her PC and follow the procedure in this section. A given card will be available as `/dev/mmcblkX` or `/dev/sdYX` where `X` is a natural number and `Y` is a letter.

It is *very important* to write to the correct device as everything will be overwritten. To avoid removing information from the wrong device, a testbed administrator can use the commands `lsblk` and/or `df -h` before and after inserting the memory card to deduce its correct device name. For our case, the device was `/dev/mmcblk0`. Once identified, to write the image to a memory card, the following command is used:

```
# dd if=${IMAGE}.img of=/dev/mmcblk0 bs=4M && sync
```

The previous `dd` and `sync` commands for copying the image to the memory card and flushing the remainder in memory to the filesystem will take tens of minutes depending on the memory card speed and the size of the image. After this is made, it is only necessary to eject the memory card and now plug it in a Raspi so it can boot up.

5. Automation and Monitoring Tools

Within the daily testbed use, there exists frequent tasks that require a set of various commands in a given Raspi. This could be tedious, prone to errors and time-consuming to realize every time the task is required to be made. Therefore, in this section, we introduce a set of tools that help to automate and monitor routinary task execution in the Raspis and show relevant example commands with them. To be able to run all the following commands, it is necessary to have SSH connectivity with the Raspis; otherwise, the commands need to be run locally on a Raspi making necessary to use a keyboard and a monitor. The testbed administrator needs to put the memory cards in the Raspis and turn them on for them to be able to boot. The devices should now be bootable.

5.1. Fabric

Controlling multiple devices using SSH from a single PC often leads to many repetitive tasks. Among these, we can mention: (i) rebooting a set of devices; (ii) installing applications in multiple devices; and (iii) copying files to/from multiple devices. Fabric [22] provides a Python library that simplifies the management of working with many devices from a single PC. First, the testbed administrator creates a directory to hold the `Fabric` source code:

```
$ export CODEDIR="${HOME}/code"
$ mkdir -p ${CODEDIR}
$ cd ${CODEDIR}
```

Then, the `${CODEDIR}/fabfile.py` file below provides a script with some basic functionalities that can perform the few items above (i–iii). In general, other administrators may require different functionalities, but this is out of the scope of this work. The following file serves as a starting boilerplate:

```
${CODEDIR}/fabfile.py
---------------------
from fabric.api import env, task, sudo
# Python Fabric script to run commands on multiple hosts through ssh
#
# Run script as 'fab <task>', where <task> is one of the scripts functions
# marked as a tesk. The task marked as 'default' will be run if <task> is not
# specified

env.hosts = ['10.0.0.100','10.0.0.101','10.0.0.102']
env.user = 'pi'
env.password = 'raspberry'

@task
def reboot():
    """ Reboot device """
    sudo('reboot', quiet=True)

@task
def install(program):
    """
    Install a program
    program: program name
    """
    result = sudo('apt-get install -y {}'.format(program), quiet=True)
    print(result)

@task
def push(src,dst):
    """
    Copy file to device
    src: source file path
    dst: destination file path
    """
    put(src, dst)
```

The previous `fabfile` shows three functions that perform our example tasks. These functions utilize variables and subsequent functions from the Fabric Application Programming Interface (API) such as `env`, `task` and `sudo` among others. Each of these API functions permits defining environment variables, creating the administrator tasks through decorators or running the mentioned task in `sudo` mode, respectively. When a task is called from the terminal, Fabric searches the directory for the `fabfile.py` file and executes the desired task. The syntax for executing a task with arguments is in the form `fab <TASK>:arg1,arg2,...` We denote the IP address of a generic Raspi for test as `<RASP_IP>`. The executions from the terminal of some of these commands are shown as follows:

```
$ fab reboot
[<RASP_IP>] Executing task 'reboot'

Done.
Disconnecting from <RASP_IP>... done.
$ fab install:tmux
[<RASP_IP>] Executing task 'install'
...
The following NEW packages will be installed:
  tmux
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
...
Preparing to unpack .../archives/tmux_1.9-6_armhf.deb ...
Unpacking tmux (1.9-6) ...
Processing triggers for man-db (2.7.0.2-5) ...
Setting up tmux (1.9-6) ...

Done.
Disconnecting from <RASP_IP>... done.
```

The first function above reboots the Raspis in the lists of hosts and the second function installs a program given by an argument. For the connection to the devices, Fabric calls the Paramiko [23] module from Python to make an SSH connection. For this to work properly, the Paramiko version needs to be higher than or equal to 1.15.1. If not available, the SSH connections from Fabric may fail. In case of any problems, some instructions for updating the Paramiko package are available in Appendix A.4 of the Appendices. This is a standard recommendation from the Fabric troubleshooting guide [24].

After a successful SSH connection is made, in the previous two commands towards the Raspi, Fabric employs the Raspi's `reboot` and `apt-get` commands in `sudo` mode to do the required tasks. Below, an example is shown for the `push` task which uses two arguments. Here, we copy `my_file` from the testbed administrator PC to a test host Raspi:

```
$ fab push:"${CODEDIR}/my_file",'~/'
[<RASP_IP>] Executing task 'push'
[<RASP_IP>] put: /home/<USER>/code/my_file -> /home/pi/my_file

Done.
Disconnecting from <RASP_IP>... done.
```

To control a large set of devices, we simply need to include them in the `env.hosts` list in the `${CODEDIR}/fabfile.py` file. Fabric has many other functionalities that are useful in controlling a large set of Raspis. For example, we may extract files or run automated experiments. The included functionalities in the `${CODEDIR}/fabfile.py` file will depend on the requirements of the testbed administrator.

5.2. Long-Running Jobs Using SSH

There are times when a task may need to run for several hours or even days on the Raspis, particularly when related to simulations or measurement campaigns. For this purpose, it might be necessary to keep open an SSH connection on the Raspis without risking that the connection will be interrupted and a given Raspi will terminate the task.

There are methods to enable the Raspis to continue running applications although the connection is terminated either on purpose or unexpectedly. One method is to run programs within a `screen` session. `screen` enables a user to run applications within a shell window, a `screen` session, which does not terminate even with connectivity interruptions. Users can attach and detach from a `screen` session as desired. The following procedure presents how to use `screen` with SSH to: (i) login to a generic Raspi; (ii) open a screen session; (iii) execute an example command; (iv) detach from the screen session; (v) terminate the SSH connection; (vi) login to the Raspi again; and (vii) attach to `screen` session to see the program still running. From the testbed administrator PC, we start by establishing an SSH connection to a Raspi and open a screen session:

```
$ ssh pi@<RASP_IP>
...
pi@<RASP_IP>'s password:

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Jul 12 13:04:31 2016

$ screen
Screen version 4.02.01 (GNU) 28-Apr-14
...
[Press space or Return to end.]
```

To enter in the `screen` session after the introduction message, we have to press either the `Space` or `Return` key in the keyboard to clear the shell. After doing so, we should be in a `screen` session although its appearance is the same as a regular terminal shell. Inside this example session, we execute a program that never ends:

```
$ top
```

The `top` command simply continuously shows the table of processes executed on the Raspi like in any Linux distribution. When `top` is running, we first press `Ctrl+a` and later `Crtl+d` in the keyboard to detach from the `screen` session. We now terminate the SSH connection and login again to verify that the `top` command is still running. Without using `screen`, the `top` program should terminate since its hosting shell was terminated. To log out, we run:

```
$ exit
logout
Connection to <RASP_IP> closed.
$ ssh pi@<RASP_IP>
```

Now that we are logged in to the Raspi again, we first check the available detached sessions by running:

```
$ screen -list
There is a screen on:
  824.pts-0.raspXX (07/12/16 13:17:30) (Detached)
1 Socket in /var/run/screen/S-pi.
```

From the command output, we can see that the session is still running in our generic Raspi number `XX` and that no user is currently attached to the session. To attach to the session, we execute:

```
$ screen -r 824.pts-0.raspXX
```

After attaching again, we should see `top` still running. `screen` has more functionalities that can be used in this or other contexts, but this is outside the scope of this work. To terminate the `screen` session, first terminate `top` by pressing `q` in the keyboard. Once `top` is terminated, we need to type `exit` two times in order to first exit the `screen` session and then terminate the SSH connection. An output should be as follows:

```
[screen is terminating]
pi@<RASP_IP>:~ $ exit
logout
Connection to <RASP_IP> closed.
$
```

6. Cross-Compilation: From the PC to the Raspberry Pi

An important case of a computational expensive task is to compile software packages and large libraries. Given the computing capabilities of the Raspi, such tasks can be challenging if not prohibitive in terms of Central Processing Unit (CPU), memory or space usage and/or compilation time. In this section, we present a procedure of how to cross-compile C++ source code from the testbed administrator PC for the ARM architecture of the Raspis. By doing this, we take advantage of the (typically) much higher computing power of the testbed administrator PC in order to save time and computational resources. Hence, we give an example of compiling a simple C++ program and copying the generated binaries with SSH to run locally on a Raspi.

Furthermore, given that our testbed purpose is for network coding applications, we also present how to cross-compile Kodo [8], a C++11 network coding library to perform encoding, decoding and recoding operations. In this way, we aim to present a fully configurable and manageable testbed with the capabilities to evaluate network coding protocols with several Raspis and locally store measurements from different evaluations. Therefore, we also show how `kodo-cpp`, a set of high-level C++ bindings for Kodo, can be cross-compiled for applications with the Raspi.

6.1. Toolchain Setup

To compile in a given architecture that is aimed for a different one, the testbed administrator needs to install a toolchain on his/her PC. The toolchain is mandatory due to the different processor architectures where the source can be compiled from. Given that compiling a toolchain can be an arduous task, we get the toolchain recommended for the ARM architecture of the Raspis. This toolchain is available from [16] and it already contains the binaries for different compilers based on `gcc 4.9`. We extract the binaries adjusting them to our coding style and compiling convention. For this, we use the `${TOOLCHAIN}` directory as the working directory. The testbed administrator may choose some other working directory of its preference if desired. First, we create the toolchain directory:

```
$ export TOOLCHAINDIR="${HOME}/toolchains"
$ mkdir -p ${TOOLCHAINDIR}
$ cd ${TOOLCHAINDIR}
```

Later, we download a Raspi toolchain with the binaries for a 64-bit Linux distribution available in [16]. Finally, we unzip the downloaded file. This is made as follows:

```
$ wget https://zenodo.org/record/154328/files/raspberry-gxx493-arm.zip
$ unzip raspberry-gxx493-arm.zip
```

Instead of calling the ARM cross compiler using its full path, we make the binaries accessible from the command shell systemwide. A way to do this is by adding the following commands in the `${HOME}/.profile` as follows:

```
$ sed -i '$a export TOOLCHAINDIR=\"$HOME/toolchains\"' ${HOME}/.profile
$ sed -i '$a export TOOLCHAINBINARY=\"raspberry-gxx49-arm-g++\"' ${HOME}/.profile
$ sed -i '$a PATH=\"\$PATH:${TOOLCHAINDIR}/arm-rpi-4.9.3-linux-gnueabihf/bin\"' ${HOME}/.profile
```

This helps the OS to recognize the location of the compiler command when a new shell is opened. The `.profile` should now contain the lines we inserted. There might be other code in the file of other testbed administrators. We recommend to leave other parts unmodified.

```
$HOME/.profile
--------------
...
export TOOLCHAINDIR="$HOME/toolchains"
export TOOLCHAINBINARY="raspberry-gxx49-arm-g++"
PATH="$PATH:${TOOLCHAINDIR}/arm-rpi-4.9.3-linux-gnueabihf/bin"
...
```

To update the `${PATH}` variable and the `.profile`, we use the `source` command for the changes take effect in the administrator system:

```
$ source ${HOME}/.profile
```

A working ARM cross-compiler in the testbed administrator PC should output the following:

```
$ ${TOOLCHAINBINARY} --version
raspberry-gxx49-arm-g++ (crosstool-NG crosstool-ng-1.22.0-88-g8460611) 4.9.3
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

6.2. Cross-Compile Example

The following shows: (i) how to cross compile the classic `hello_world` C++ example for the Raspi ARM architecture and (ii) how to copy and execute the binary in a Raspi using Secure Copy (SCP) and SSH. First, we create the file `hello_world.cpp`. For simplicity, we create it in the directory where we stored the `fabfile.py` file with the following content using any text editor:

```
${CODEDIR}/hello_world.cpp
--------------------------
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

We save the previous file and compile it for Raspi in the testbed administrator PC by doing:

```
$ ${TOOLCHAINBINARY} hello_world.cpp -o hello_world
```

This should produce a binary `hello_world` that is executable on the Raspi. We copy it to a Raspi using SCP and using Fabric instead if we are interested in deploying a compiled binary for many Raspis.

```
$ scp hello_world pi@<RASP_IP>:~/
```

After the executable has been copied to the Raspi, we login through SSH to it:

```
$ ssh pi:<RASP_IP>
```

We can list the directory content after we have logged into the Raspi and verify that the compiled `hello_world` binary is there:

```
pi@<RASP_IP>:~ $ ls
hello_world  rasp_config
```

Finally, we simply execute the `hello_world` to confirm that the cross-compiling of `hello_world` worked properly:

```
pi@<RASP_IP>:~ $ ./hello_world
Hello World!
```

6.3. Cross-Compile Kodo

As we originally mentioned, Kodo is a C++11 network coding library that permits implementation of network coding functionalities by allowing any network protocol designer to use and test the primitive encoding, decoding and recoding operations of RLNC. In this way, a designer only needs to focus on the design and test of a network coding-based protocol. Kodo is available through programming bindings for a variety of popular programming languages. This procedure will present how to configure the Kodo C++ bindings `kodo-cpp` to cross-compile applications that can run in Raspi. `kodo-cpp` provides a simple interface to the underlying C++11 code that exists in the libraries `kodo-core` for the object structure and `kodo-rlnc` for the RLNC codec implementation. More details about Kodo are provided in the code documentation [25].

To use Kodo for research, it is necessary to obtain a research free license. To do this, a request form needs to be filled in [26] and wait for it to be processed by the Kodo developers. Once the access for Kodo has been granted, the source code can be pulled from its Git repositories to be compiled. Assuming that the testbed administrator already has access, we clone the `kodo-cpp` repository locally in `$CODEDIR` and change directory into the repository by doing:

```
$ cd ${CODEDIR}
$ git clone git@github.com:steinwurf/kodo-cpp.git
$ cd kodo-cpp
```

We first configure `kodo-cpp` to build executables for the ARM architecture using the Raspi toolchain and later build them by running:

```
$ python waf configure --cxx_mkspec=cxx_raspberry_gxx49_arm
...
'configure' finished successfully (X.XXXs)
$ python waf build
...
'build' finished successfully (XmXX.XXs)
```

If the configuration and build steps are successful, the binaries should have been created. To be able to use them, we need to create a shared library that we will use in the Raspi. To do this, we run the following command:

```
$ python waf install --install_shared_libs --install_path="./shared_test"
...
'install' finished successfully (X.XXXs)
```

Now, we copy the shared library, binary files and related headers to the Raspi home directory as follows:

```
$ scp -r shared_test/include shared_test/libkodoc.so pi@<RASP_IP>:~/
```

Alternatively, and for the testbed administrator reference, Kodo can also generate static libraries. We log in to the Raspi and execute the unit tests and one of the binaries by running:

```
$ ssh pi@<RASP_IP>
$ ./kodocpp_tests
...
[  PASSED  ] X tests.
$ ./encode_decode_simple
Data decoded correctly
```

If the Kodo cross-compilation worked properly, both the unit tests and binaries run should provide the shown outputs.

7. Conclusions

Observing the expectation of the IoT and lack for a low-cost, easy-to-configure testbed in this area for reproducible research, we provide an in-depth description of the new Aalborg University's Raspi testbed for network coding evaluation and how to guarantee replicability and scaling management of this system. The description shows how to set up interconnected Raspis with memory cards for local storage, a Raspbian Lite image, network connectivity and proper system administration privileges. Using the presented procedure permits setting up a Raspbian Lite image for the Raspis. A tailored Linux distribution might be created from the scratch using the Yocto project. However, to assemble and compile the software for the Raspi can be a tedious and time-consuming task. However, this method could be adequate for an expert user. We hope this work permits researchers to replicate setups and scenarios for evaluating their strategies in a rapid and manageable way. Future work in the use of Raspi devices will focus on expanding the setup and automation of tasks to run the testbed, configure specified network topologies (e.g., with specific connectivity or packet loss ratios), reserve the use of these sub-networks for running tailored experiments and open the use of the testbed beyond our team at Aalborg University. Future work in this area will consider making the testbed fetch the image through the HTTP server. This is expected to simplify the maintenance of the memory cards.

Acknowledgments: This research has been partially financed by the Marie Curie Initial Training Network (ITN) CROSSFIRE project (Grant No. EU-FP7-CROSSFIRE-317126) from the European Comission FP7 framework, the Green Mobile Cloud project (Grant No. DFF-0602-01372B), the TuneSCode project (Grant No. DFF-1335-00125) both granted by the Danish Council for Independent Research (Det Frie Forskningsråd), and by the German Research Foundation (DFG) within the Collaborative Research Center SFB 912—HAEC.

Author Contributions: The authors contributed equally to this work.

Conflicts of Interest: The authors declare no conflict of interest. The founding sponsors had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, and in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

API	Application Programming Interface
ARM	Advanced RISC Machine
Bash	Bourne Again SHell
CPU	Central Processing Unit
DHCP	Dynamic Host Configuration Protocol
HTTP	Hyper Text Transfer Protocol
IoT	Internet of Things
IP	Internet Protocol
IT	Information Technology
LAN	Local Area Network
MAC	Medium Access Control
NC	Network Coding
NFS	Network File System
OS	Operating System
PC	Personal Computer
QEMU	Quick Emulator
RAM	Random Access Memory
Raspi	Raspberry Pi
RLNC	Random Linear Network Coding
SCP	Secure Copy
SD	Secure Digital
SSH	Secure Shell
URL	Uniform Resource Locator

Appendix A. Alternative Commands for Outdated Packages

This section describes alternative commands in case the testbed administrator is not able to update old packages on his/her Linux distribution for performing the commands, particularly the ones regarding the `util-linux` package or if a command just fails.

Appendix A.1. Losetup for Loopback Devices

If the `losetup -P` command shows the `invalid option` message, an alternative is to manually set all the loopback devices used during the whole procedure with `mknod`. To do this, the alternative commands are:

```
$ export DEV="/dev/loop0"
# mknod ${DEV}p1 b 7 1
# mknod ${DEV}p2 b 7 2
# losetup -o $((8192*512)) --sizelimit $(( (137215-8192+1)*512) )) ${DEV}p1 ${IMAGE}.img
# losetup -o $((137216*512)) --sizelimit $(( (4806655-137216+1)*512) )) ${DEV}p2 ${IMAGE}.img
```

However, there will be a few differences by using the above code as an alternative to the `losetup -P` case. First, the output from `lsblk` will look different:

```
# lsblsk
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
...
loop1    7:1    0    67M  0 loop
loop2    7:2    0   2.2G  0 loop
...
```

Second, detaching the loop devices will also be different:

```
# losetup -d ${DEV}p{1,2}
```

Working with these loopback devices will be transparent for the remainding procedure.

In addition, when creating the persistent home directory partition in Section 4.2, for recreating the loopback devices, the commands are:

```
$ export DEV="/dev/loop0"
# mknod ${DEV}p2 b 7 2
# mknod ${DEV}p3 b 7 3
# losetup -o $((137216*512)) --sizelimit $(( (4806655-137216+1)*512) ) ${DEV}p2 ${IMAGE}.img
# losetup -o $((4806656*512)) --sizelimit $(( (6903807-4806656+1)*512)) ${DEV}p3 ${IMAGE}.img
```

Finally, for detaching in this case, the command is:

```
# losetup -d ${DEV}p{2,3}
```

Appendix A.2. Image Chroot with Proot

In some cases, `chroot` may not work properly. In this case, an alternative can be `proot`. It might be required to install it before changing the root with `apt-get install proot`. Then, it should be possible for the testbed administrator to run the following:

```
# proot -q qemu-arm-static -S ${ROOTDIR}
```

Appendix A.3. Umount after Image Chroot

In case the `umount --recursive` command shows the `invalid option` message, an alternative is to manually unmount all the partitions used during the `chroot` environment in the reverse order from which they were mounted. These commands are:

```
# umount ${ROOTDIR}/dev/pts
# umount ${ROOTDIR}/dev
# umount ${ROOTDIR}/sys
# umount ${ROOTDIR}/proc
```

Appendix A.4. Paramiko Package Update

In case Paramiko version 1.15.1 or higher is not installed, we may observe a key exchange algorithm error when trying to log in through SSH to a Raspi with Open SSH. In this case, it might be necessary to update the Paramiko package that Fabric uses to remove this error by running:

```
# pip install --upgrade paramiko
...
Successfully installed cffi-1.7.0 cryptography-1.4 paramiko-2.0.1
# pip show paramiko
...
Name: paramiko
Version: 2.0.1
...
```

References

1. Evans, D. *The Internet of Things: How the next evolution of the Internet is changing everything;* Cisco Systems Inc.: San Jose, CA, USA, 2011.

2. Ahlswede, R.; Cai, N.; Li, S.Y.; Yeung, R.W. Network Information Flow. *IEEE Trans. Inf. Theory* **2000**, *46*, 1204–1216.
3. Koetter, R.; Médard, M. An Algebraic Approach to Network Coding. *IEEE/ACM Trans. Netw.* **2003**, *11*, 782–795.
4. Ho, T.; Médard, M.; Koetter, R.; Karger, D.R.; Effros, M.; Shi, J.; Leong, B. A Random Linear Network Coding Approach to Multicast. *IEEE Trans. Inf. Theory* **2006**, *52*, 4413–4430.
5. Pahlevani, P.; Lucani, D.E.; Pedersen, M.V.; Fitzek, F.H. Playncool: Opportunistic network coding for local optimization of routing in wireless mesh networks. In Proceedings of the 2013 IEEE Globecom Workshops (GC Wkshps), Atlanta, GA, USA, 9–13 December 2013; pp. 812–817.
6. Szabo, D.; Gulyas, A.; Fitzek, F.H.P.; Fitzek, F.H.P.; Lucani, D.E. Towards the Tactile Internet: Decreasing Communication Latency with Network Coding and Software Defined Networking. In Proceedings of the 21th European Wireless Conference European Wireless 2015, Budapest, Hungary, 20–22 May 2015; pp. 1–6.
7. Lucani, D.E.; Pedersen, M.V.; Heide, J.; Fitzek, F.H.P. Fulcrum Network Codes: A Code for Fluid Allocation of Complexity. 2014, arXiv:1404.6620. arXiv.org e-Print archive. Available online: https://arxiv.org/abs/1404.6620 (accessed on 10 October 2016).
8. Pedersen, M.; Heide, J.; Fitzek, F. Kodo: An Open and Research Oriented Network Coding Library. In *International Conference on Research in Networking*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6827, pp. 145–152.
9. Cox, S.J.; Cox, J.T.; Boardman, R.P.; Johnston, S.J.; Scott, M.; O'Brien, N.S. Iridis-pi: A low-cost, compact demonstration cluster. *Clust. Comput.* **2014**, *17*, 349–358.
10. Raspberry Pi Foundation. The Making of Pi. Available online: https://www.raspberrypi.org/about (accessed on 10 October 2016).
11. Leccese, F.; Cagnetti, M.; Trinca, D. A Smart City Application: A Fully Controlled Street Lighting Isle Based on Raspberry-Pi Card, a ZigBee Sensor Network and WiMAX. *Sensors* **2014**, *14*, 24408–24424.
12. ARDUINO Corp. Available online: http://arduino.cc (accessed on 10 October 2016).
13. Leccese, F.; Cagnetti, M.; Calogero, A.; Trinca, D.; di Pasquale, S.; Giarnetti, S.; Cozzella, L. A New Acquisition and Imaging System for Environmental Measurements: An Experience on the Italian Cultural Heritage. *Sensors* **2014**, *14*, 9290–9312.
14. Sapes, J.; Solsona, F. FingerScanner: Embedding a Fingerprint Scanner in a Raspberry Pi. *Sensors* **2016**, *16*, 220, doi:10.3390/s16020220.
15. Predictable Network Interface Names. Available online: https://www.freedesktop.org/wiki/Software/systemd/PredictableNetworkInterfaceNames (accessed on 10 October 2016).
16. Configuration Files and Scripts. doi:10.5281/zenodo.154328. Available online: https://doi.org/10.5281/zenodo.154328 (accessed on 10 October 2016).
17. The Yocto Project. Available online: https://www.yoctoproject.org (accessed on 10 October 2016).
18. The Raspbian Distro. Available online: https://www.raspbian.org (accessed on 10 October 2016).
19. Qemu User Emulation. Available online: https://wiki.debian.org/QemuUserEmulation (accessed on 10 October 2016).
20. GNU Screen. Available online: https://www.gnu.org/software/screen (accessed on 10 October 2016).
21. Overlayroot. Available online: https://github.com/chesty/overlayroot (accessed on 10 October 2016).
22. Fabric Documentation. Available online: http://www.fabfile.org (accessed on 10 October 2016).
23. Python Paramiko. Available online: http://www.paramiko.org (accessed on 10 October 2016).
24. Fabric Troubleshooting Guide. Available online: http://www.fabfile.org/troubleshooting.html (accessed on 10 October 2016).
25. Kodo-cpp Documentation. Available online: http://docs.steinwurf.com/kodo/kodo-cpp/index.html (accessed on 10 October 2016).
26. Steinwurf Research License webpage. Available online: http://steinwurf.com/license (accessed on 10 October 2016).