SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's thesis in Informatics

Multi-Tenant Edge Testbed using Heterogenous Commodity Hardware

Markus Alexander Gruhlke

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's thesis in Informatics

Multi-Tenant Edge Testbed using Heterogenous Commodity Hardware

Mehrbenutzer Edge Testumgebung auf Basis von heterogener handelsüblicher Hardware

Author:	Markus Alexander Gruhlke
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	Dr. Nitinder Mohan
Submission Date:	15.11.2023

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.11.2023 Markus Alexander Gruhlke

Acknowledgments

I would like to thank my advisors, Dr. Nitinder Mohan and Raphael Hetzel for their incredible patience with me during this thesis. Without their continued support and engagement, I would have been unable to complete the work. I would also like to express my gratitude to Prof. Dr. Jörg Ott and the Chair of Mobility for providing me with the opportunity to write my thesis there. Finally, I wish to thank my parents, as without their support, I would never have been able to study at TUM.

Abstract

Replication is at the core of scientific research. The scientific method relies on researchers confirming published results by repeating the experiments. In the field of edge computing, testing and experimenting are associated with significant additional work. Distributed systems are already difficult to handle, with experiments focusing more on how components interact with one another than any particular results of the computation. The edge adds the challenge of handling physical devices to these problems. While cloud applications may use simulation to mimic the behavior of the real system, the edge is by definition restricted to its devices. Many applications on the edge even rely on the availability of low-level hardware interactions. Therefore, experiments need to run on actual edge hardware as well to ensure a realistic and workable environment. To facilitate the use of real hardware, so-called testbeds are developed. A testbed assists the users in allocating and managing hardware resources while administering access without manual interference.

This thesis discusses the development and implementation of such an edge testbed. The core focus of the testbed is adaptability. Both edge and edge devices have rather ambiguous definitions and include a wide range of possible applications and hardware. Therefore, the testbed needs to be adaptable; new and different devices may have to be added to the system, and various experiments have to be supported by it. The goal is to develop a testbed that gives users the freedom to interact with the hardware however they desire. The thesis describes the design and implementing considerations that went into the development.

Contents

1. Introduction

The *Edge* and *Edge Computing* are two broad terms that exist somewhere between the *Internet of Things*, *Cloud Computing* and *Distributed Computing*. An exact definition of the topic is difficult, as the projects that are part of the field range from small servers with dedicated AI capabilities[LLG19] to repurposed routers [LWB16]. Generally, edge computing can be described as an attempt to provide the services associated with cloud computing in a more localized manner [Cao+20].

Since the field is so broad, many different approaches can exist, giving a wide range of potential developments, and with many developments, testing becomes an important issue. Testing and experimenting with distributed systems is already a difficult problem [IT18]. Running an experiment that relies on the interaction between multiple devices is generally more complicated than running an experiment on a single machine. That is why emulating platforms like *emulab*[1] are used to simulate a real network with virtualization rather than physically recreate it. This is not a reasonable alternative for working on edge systems. Edge systems are defined by their limited capabilities, their differing architecture or other peculiarities. This does not mean edge devices have to be weak; they can have the GPU cores needed to do complicated parallel calculations, but unlike the cloud, they are not datacenters filled with resources. All this is to say that the edge relies on its hardware, and therefore it is only reasonable to also experiment and test with actual hardware typical of the field.

With handling hardware becoming a necessity for successful edge research, the problems associated with doing so need to be handled reasonably. It cannot be the case that a researcher is required to plug the devices together manually and install operating systems on the devices with a USB stick. Testbeds are an attempt to lessen this burden. The Testbed would allow users to access the hardware resources, reserve resources for their own experiments, and get help installing and configuring the provided hardware.

Experimenting is also relevant for the replication of studies and results. If a researcher writing their own paper has to surmount these problems, there is little incentive to recreate any experiment, and with widely different hardware available, it may not even be possible. This is another use case for testbeds. They not only make rerunning an experiment trivial; they also automatically document experiment setups. Any

[1] `https://www.emulab.net/`

description of an experiment that can be used by the testbed to create it can also be used by other researchers to recreate it.

1.1. Structure of the Thesis

The thesis starts in chapter 2 with the problems of replication and the origin of the overall discussion. Followed by an outlook on existing testbeds and the wide variety of goals and approaches to the concept. Looking both at systems more associated with the cloud and more focused on edge devices.

In the main part, the thesis focuses on the design, implementation and evaluation of the developed testbed. Starting in chapter 3 with the goals of this particular system, describing the overall workflow and the components of the testbed. Then, in chapter 4, going into more details of the components and adapting the points developed in the design into a workable prototype. Finally, evaluating if the testbed is functional and the performance of the major steps in the process is discussed in chapter 5.

In the final chapter 6, a conclusion of the project is given with an outlook on what parts are finished and what still needs to be developed or implemented.

2. Background & Related Work

Testbeds are an attempted solution to what is referred to as the "Replication Crisis". A meta-scientific term first coined in the article *Is the Replicability Crisis Overblown? Three Arguments Examined* [PH12]. The term tries to encapsulate a discussion that has been growing since the 2000s about the reproducibility of scientific studies. The discussion started in the fields of psychology and medicine with papers like *Why Most Published Research Findings Are False* [Ioa05] by John P. A. Ioannidis trying to quantize the false-positive rate of scientific studies. From there, the discussion spread to other scientific disciplines. In 2016 Nature[1] published a survey in which most participants from the fields of chemistry, biology and physics responded that they had failed to reproduce experiments [Bak16]. The article also mentions that only a minority of researchers have ever attempted to publish a replication study. Citing bad incentives for positive replication and the reluctance of journals to publish negative results as potential reasons.

In *Reproducibility of Scientific Results* [FW21] the authors try to define the crisis by separating it into five points. Firstly, they also mention the two already described characteristics of the replication crisis. Naming the "absence" of both positive and negative replication studies, and the "publication bias" towards positive results. They add the failure of large-scale replication attempts, in which researchers attempted to reproduce a wide range of studies published in the previous years, and "questionable research practices" of authors selecting confirming measures to publish a successful paper rather than discard the research as additional parts of the overall problem. These four issues have a more societal and institutional layer. They require changes in incentive and the general approach to the scientific process. The final point focuses more on the technical issues with execution and documentation of the experiments, where the authors lament missing "method, data and analysis" in scientific papers needed to replicate the results. Unlike the societal problems, this could be solved individually by better standardization of experiment layout and recording.

All these papers are part of fields outside the computer science. But the problem and debate has also reached there. In *Reproducibility in Scientific Computing* [IT18] the authors analysis the extends of the problem for the computer science. They remark that computer science are uniquely capable of being reproducible. Unlike most other

[1] `https://www.nature.com`

natural sciences computer science control nearly every variable in an experiment. Still there exist a wide variety of environmental factors that can impact the result, some being rather unexpected factors. Using a simple terminal command as an example they separate the environmental factors into different layers. Obviously the command implementation and input data impact the result, but also the used underlying software and versions. The functionally same program implemented in java or python could lead to execution from the perspective of the hardware and therefore impact the overall result. Below the software layer operating system and kernel versions could effect the computation in the same way as software. The lowest level are the hardware differences. Unlike with the other layers replicating hardware is not easily possible. Replicating hardware could be interpreted as using the exact same device, the same type of device or a device with the same capabilities.

The authors conclude that there are solutions for replicating most computation tasks, especially for computations on a singular device. But they remark that this is not the case for distributed systems, with more complex workflows. They especially mention the added complexity through heterogenous hardware. Finally they discuss, that while frameworks supporting researches running experiments exists, additional tools are required to allow scientists to focus on their domains instead of how to run experiments. But they also mention the needed change in approach similar to the discussion in other disciplines.

2.1. Existing Testbeds

One such type of framework potentially assisting researches are testbeds. They find a lot of use in the field of cloud computing. One such example is *CloudLab* [RET14; Dup+19]. CloudLab is a multi tenant cloud system, that allows users to allocate hardware resources for their experiments, acting similar to commercial cloud providers. They differ in the ability of CloudLab to extend beyond building applications on the cloud to, building cloud infrastructure as part of the experiment itself.

Their infrastructure consists of three sites each with a different approach to cloud computing. The first uses a combination of x86-64 and low-powered ARM servers to allow for experiments using differing levels of power. The second site representing traditional datacenters with 4000 cores for computations. Finally, the third site focuses on storage and virtual machines to be connected to outside experiments. All sites utilize powerful network hardware with each device being connected with two 10Gb/s networks.

Users reserve and configure infrastructure using profiles. A profile consists of a list of the requested physical hardware as well as the software components needed to run the experiment. The profile concept allows for reuse and public sharing of configurations. Other researchers can then easily replicate the experiments or adapt the profiles for their own needs.

Since its release in 2014 CloudLab has been used in thousands of experiments and the authors analyzed the results in 2019 [Dup+19]. One of their observations was that many experiments used CloudLab because of the low level access to the hardware, allowing for configuration on a level impossible in virtualized hardware, and the isolation achieved through the slicing concept, giving users complete control over their network.

The characteristic point of CloudLab is the ability to experiment with the control functions of a datacenter itself. The *Chameleon Testbed* [Kea+20] focuses more on providing resources for large scale experiments. In doing so it bridges the gap between cloud and edge testbeds. It is mainly made up of powerful datacenter-grade hardware, but extended with special clusters for GPU heavy experiments, FPGAs and other low-powered edge devices. Similar to CloudLab users are able to configure their own sub-networks using OpenFlow [Hal+15]. The testbed allows for bare-metal configuration by installing "whole-disk images" onto the nodes using PXE and iSCSI. This also leads to an separation between the data and control flows. Nodes can communicate via the ethernet interface and be programmed via the SCSI. As such experiments in the Chameleon Testbed can be completely redeployed on a single lease, both the network and the nodes can be changed during the experiment.

Both CloudLab and Chameleon mention, in their analyses of their testbed, that adaptability is an important characteristic of the testbed. Citing their associated surveys of user behavior, they claim that users chose their testbed because they would implement new, in-demand technologies.

A different approach is used by the *ORBIT Testbed* [Ott+05; Ray+05]. Instead of bare-metal installations, a small "Node Agent" stays in control of the device running the experiment software. ORBIT focuses on wireless networks, trying to recreate realistic network conditions by having grids of devices communicate with one another. It is further extended by additional special-purpose wireless configurations, for example, moving nodes. Staying in control of the hardware allows for a more standardized approach to experiments; instead of giving the users access to the hardware, the users interact with the hardware via configurations that the testbed then executes. This can be seen in the layout of the ORBIT testbed in figure 2.1 (a). The user interface is only a minor part of the entire system and can only communicate with the "Experiment Controller". The controller then instructs the other components to install software, run the experiment and collect data. The same can be seen from the lifecycle in figure 2.1 (b), the user only provides the initial script to be executed; every step after this is part of the system. The controller sends information to the handler, the handler records

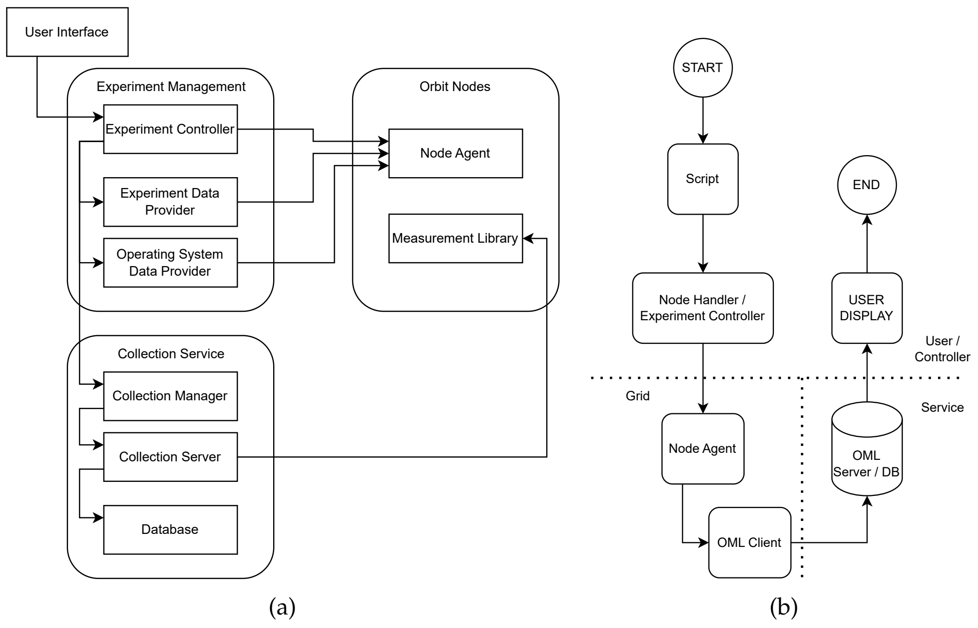


Figure 2.1.: Layout (a) and Lifecycle (b) of the ORBIT Testbed

information via the Orbit Measurement Library (OML) and the OML then reports the information back to the user.

An ORBIT experiment description is made up of five sections. The *topology* defines the used hardware consisting of the devices as well as the links between the devices. Next, the *application* section includes a list of used software components. ORBIT tries to reuse the applications through a shared repository. An application would then have a single purpose, like traffic sources and sinks. The *prototype* section then specifies the nodes by defining the applications and parameters installed. These specified node prototypes are then mapped to the booked devices in the *mapping* section. Mapping can be done via strategy, either with a simple list or stochastically. Finally, the *staging* selection allows for changing parameters during the experiment. The reduced freedom means that ORBIT can be more descriptive in their experiment process. Users can focus on describing what their experiment should look like instead of how it needs to act. Initial configuration can be done by defining properties, and recording the measurement can be done via the ORBIT Measurement Framework by defining measurement points.

ORBIT comes with many restrictions on the experiments, but in turn, it can assist users more during these experiments.

Another different approach is the *i8-Testbed* [HJG20]. Unlike the other systems, it is incapable of dynamically changing its network infrastructure. Instead, it provides small,

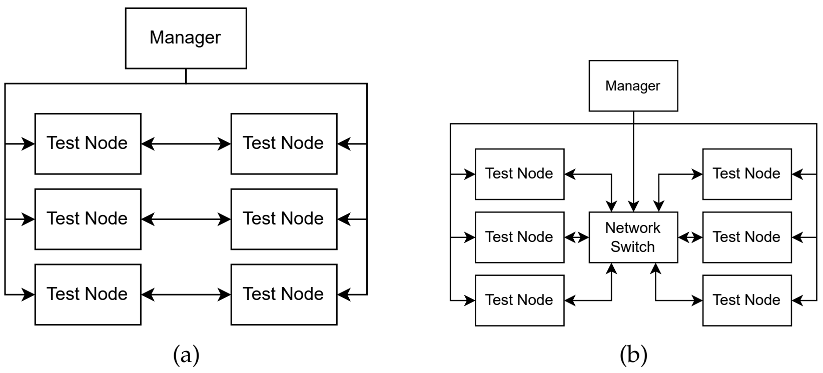


Figure 2.2.: Physical configurations used by the i8-testbed

pre-configured clusters with a management node and multiple test nodes. Figure 2.2 shows two such configurations described in the paper. The group of simple server pairs in (a) can, for example, be used to evaluate a firewall policy and how it affects transfer rates, while the interconnected nodes in (b) could be used to experiment with *Software Defined Networking* protocols. Each individual configuration serves its own special purpose, and while the exact configuration changes, the general layout with manager handling each device individually stays the same. It also shows the clear separation between data and control network. The manager has its own network connections to every device; even the network switch in (b) is controlled by the manager.

Also, the testbed does not install software onto the devices, instead using live images and starting from a clean state for every test run. The manager installs the software, runs the experiment and collects the data. This leads to a very fixed experiment workflow, which has the advantage of being easily configurable, comparable to ORBIT, but loses the possibility for different experiments, both for changing hardware and differing experiment designs.

All in all, there exists a wide variety of testbed approaches somewhere between the fields of cloud and edge computing. Depending on the focus points of the testbed this leads to varying restrictions on the experiments and to testbeds of widely different scales.

3. Design

The core idea behind any testbed is to assist users and administrators in managing, utilizing, and allocating the available hardware resources.

This basic description includes the core requirements that a testbed should fulfill. The testbed must allow users to interact with the hardware resources in some way. In an edge testbed the used devices are rather limited and cannot provide full virtualization, instead full access to the functionality of the hardware is provided. Additionally, edge applications often interact with the hardware resources on a low-level, requiring unrestricted access to the complete system resources. To facilitate this access, the testbed needs to be capable of installing operating systems directly onto the devices. Installation is a labor-intensive step in the process of running an experiment, and it is required for every single experiment. Therefore, automating this step is a significant help when interacting with the hardware.

In managing the hardware, the testbed should be in control of the resources. Without outside interference, the testbed needs to be capable of returning to an initial base state from which to start the experiments. Without the ability to clear the system, the testbed cannot allocate resources to following users.

Finally, the testbed should "assist" the users. It should support them and make it easier to run their experiments. Most relevantly, it should not set up additional barriers to what kinds of experiments can be done and how the users can interact with their experiment. Users may require remote access to their experiments, or the experiments may need to interact with components outside the testbed via the internet. The testbed should give users the freedom to run experiments however they want and trust that users will not abuse such freedoms.

The Edge-Testbed adds two further considerations to these basic principles. Firstly, the testbed is "Multi-Tenant", meaning that multiple experiments may run concurrently and share some of the network resources. This should allow for more efficient use of the available hardware, especially in smaller experiments where locking a significant amount of hardware results in either far fewer experiments being possible at the same time or far more requirements for the network hardware. If every experiment blocks all the used network nodes, it would require far more redundant links and alternative routes to still facilitate the same number of experiments concurrently. Otherwise, two nodes that are distant from each other in the network could potentially require all the available network nodes. While resource sharing gives the testbed more versatility, it also results in non-interference being a significant requirement to still guarantee that experiments are run in isolation from one another. If experiments could interfere, the scientific results of any experiment are questionable, as the same experiment could produce different results depending on the load of the testbed.

Secondly, the Edge-Testbed focuses on “Commodity Hardware”. The main idea behind “Commodity Hardware” is that experiments require wildly different types of hardware, from single-board computers up to dedicated GPUs with AI capabilities. Especially edge systems are often defined by their limited capabilities, and the testbed should reflect these limitations in its hardware choices. This means the testbed should not put stringent requirements on both the network devices and the hardware nodes running the experiment. One example would be remote management interfaces. Some devices may support such a system, and in that case, the testbed may use the available functionality, but it should not be a requirement for all hardware to support such a system. The system should therefore not require certain hardware with specific capabilities but abstract the different hardware down to its most basic functionality and then dynamically use the correct processes and available support technologies that come with each individual hardware component.

Additionally, the flexibility of the testbed makes it possible for the implementor to use whatever hardware they have available instead of requiring specific server-grade devices.

3.1. Shared Hardware

There are two types of resources managed by the testbed: end nodes and the network.

The end nodes are not shared between experiments. This gives the experiment maximal control over the devices and therefore more freedom in what kind of experiments can be run on the testbed. Furthermore, the requirement for “Commodity Hardware” means that many devices may not even be capable of running a virtualization environment, severely limiting what devices can be used as part of the testbed.

Future expansion with more server-grade hardware with virtualization and shared nodes would still be possible, but it is not a focus point of the design. For example, isolation would need to be considered more rigorously, as experiments would share the same network card, leading to potential interference. Such an expansion could be useful to run controllers for the experiments, as such an application would not be part of the measured communication while not blocking an additional node for administrative purposes.

Network resources, on the other hand, are shared in every way. Both network devices, like switches and network links connecting these switches may be shared between experiments. These limited restrictions lead to far weaker network guarantees than exclusive use of the available hardware. Instead, the guarantees rely on the capabilities of VLANs to isolate traffic from other experiments. By having weak network guarantees, there are fewer restrictions on the hardware itself. A separate management and experiment plane would require all hardware to have multiple network cards, which would be incompatible with using commodity hardware as part of the testbed.

3.2. Running Experiments

An experiment can be separated into five distinct steps. In the **scheduling** step the user needs to provide the testbed with information about the experiment they wish to run. Most relevantly, users need to define the required hardware, the needed bandwidth for the experiment and the expected length of the experiment. All these information are required for the testbed to decide, if the experiment can be run on the available infrastructure at all and after that finding a timeslot to run the experiment. The user may then book the provided timeslot, which reserves the hardware for their exclusive use for the requested time. The user may select their hardware either specifically using an identifier to select one specific device or a shared characteristic between devices in which case the testbed decides which specific device it allocates for the experiment. For example the system has multiple of the same type of single-board-computer, the user may not care which one of these is used for the experiment only that the device has the expected capabilities. In such a case the system can choose the specific device based on network conditions, while the user does not have to consider the actual layout of the physical infrastructure. In both cases the specific devices used by the experiment will be fixed at the end of scheduling. This prohibits rescheduling an experiment at a later time, but it allows the user to rely on information like IP- or MAC-addresses about the specific hardware in their software.

All the information provided by the user during the scheduling step are recorded into the experiment descriptor. The descriptor allows for reproducing an experiment by storing all the decisions made by both user and testbed during the scheduling step making them repeatable in future reruns.

Once the scheduled time is reached the testbed will start the **installation** step. While the os images do not need to be provided immediately with scheduling at this point the information needs to be present, or the experiment will be cancelled. Also there needs to exist a mapping to tell the testbed, which image to install onto which devices.

The testbed handles the installation process automatically. As part of the flexibility of the testbed it first dynamically loads the correct way to interact with the device and then installs each device in a distributed manner. Since installation is a time-consuming step it can be parallelized by running multiple installing instances. Finally, the system will configure the selected network devices to separate the experiment into a virtual network and start all the devices.

After this, the experiment is **running**. During this step the testbed is not interacting with the experiment at all to ensure that there is no interference. That does not mean the experiment is completely encapsulated. The user may still interact with the nodes using for example SSH or nodes may still reach out to the internet, this could lead to interference and skew results. However it is more useful to allow the user to do something, even if it could break an experiment, rather than not allow it and arbitrarily limit the possible experiments. The isolation during the running phase of the experiment is only in regard to other testbed functions or other experiments.

One way the testbed could still send messages to the experiment are pre-scheduled messages. For example, the experiment may need to switch to a different set of test parameters after some time. The user could implement such functionality on their own, but since the functionality could be used by multiple experiments simultaneously and there are no real interference concerns, a central function provides an easier access for the user and standardizes behavior.

Next comes the **collection** step. Data collection falls into the responsibility of the experiment. The testbed is not actively selecting data for storage, instead it provides a system for the experiment nodes to send their data to. The data is then stored and may be downloaded by the user at a later time. Of course individual approaches ignoring the testbed are also possible within capabilities given to the experiment. For example, data could simply be sent to a cloud service outside of the testbed.

The running and collection step is not mutually exclusive from the perspective of the testbed. It is only a reasonable design paradigm, based on the need to alternate between running the experiment and sending collected data. Depending on the experiment it may be useful to regularly upload the collected data. As there is no difference between the steps, it is the experiment's responsibility to start the collecting process before the end of the timeslot. The testbed again can provide support for starting collection on time via the pre-scheduled messages, but alternatives like a simple timer could also be sufficient.

Both the running and the collecting step are handled very broadly by the testbed. This is to ensure maximal freedom for the experiments. The testbed provides supporting functionality, but does not make its use mandatory. This leads to a two layered approach. At the core the system allocates resources to a user, installs operating system images, starts and stops devices. To facilitate easier experiments the testbed can collect data and send messages to the experiments.

Finally, the experiment reaches the end of its reserved timeslot and enters the **clean-**

up step. The system uses a very limited clean-up process. Devices are forcefully shutdown and the created virtual network is removed. There is no deinstallation of the operating images as the next experiment simply overwrites the installed image. As such all information remaining on the devices may be considered lost but not deleted. Secure information could be accessed by other experiments, and therefore it may be in the users interest to delete such information before the timeslot ends.

3.3. Core Components

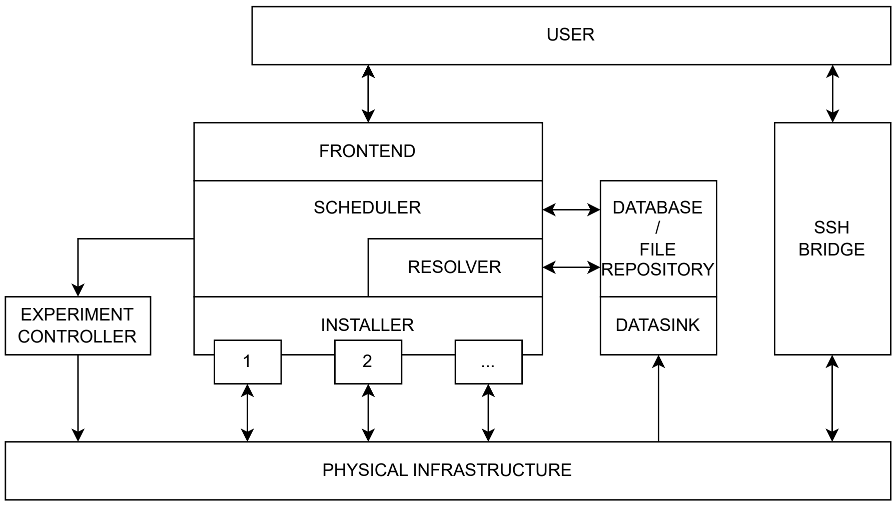


Figure 3.1.: The different components of the Edge-Testbed

Figure 3.1 shows all the individual parts of the Edge-Testbed. The components of the testbed connect the user and the physical infrastructure. It is made up of the three mayor components **scheduler**, **frontend** and **installer**. As well as multiple supportive components that, either provide optional functionality for the experiments, this includes the datasink, the ssh bridge and the experiment controller, or they provide internal services to other components of the testbed like the data storage and resolver.

Scheduler, frontend and installer form the mayor components because they are central to the installation functionality of the testbed. The Frontend collects all the information during the scheduling step and passes it to the scheduler for the scheduling decision. Once the scheduler finds a valid timeslot the frontend presents the information to the user for confirmation. On reaching the booked timeslot the scheduler starts the installation process by sending the data to the installers. At the end the scheduler also starts the clean-up process again through the installers.

3.3.1. Scheduler

The scheduler is the core component of the testbed. It is the decision-making component of the system. All other components either provide information to the scheduler or are instructed by the scheduler.

The main decision the scheduler needs to handle is calculating if an experiment can be run on the testbed and if so when the experiment can be run. One of the goals behind the testbed is abstracting the complicated network reality by hiding the details of the physical setup from the users. The user requests the end devices and network conditions from the testbed, and it is up to the testbed to map these **requirements** to the physical infrastructure.

The experiment requirements consist of the requested hardware, requested bandwidth and expected length of the experiment. Hardware may be requested either by a unique identifier or by a class identifier. For the most part users will not be interested in the specific device that runs their experiment, but rather that the device supports a certain format of OS images or has certain capabilities like a dedicated GPU. Still specific selections need to be included for example for reruns of the exact same configuration. Also, there is no guarantee that the same configuration will lead to the same network every time even with specific device selection. While it can be generally assumed that the network is rather static, administrators may alter the network structure by switching, adding and removing nodes at any time. The testbed removes the network specifics from the purview of the users to reduce the complexity of running an experiment, but this also limits the user in controlling all variables of the experiment.

The bandwidth and length of the experiment need to be estimated by the user and both should be a generous overestimation of the expected value. While this potentially wastes the available resources, underestimating the required bandwidth could impact the result of the experiment and underestimating the time could lead to the collection step being interrupted by the testbed ruining the result. Especially the installation time needs to be considered for the estimation, since the system starts the installation at the scheduled time, it is up to the user to include this in their estimation. Estimating these measures places a significant a priori burden on the user, who needs to have a good understanding of what their experiment will do and how it will behave on the hardware. For example, a simple file transfer task will not have an inherent length, its length depends on speed and file size and speed is a characteristic of the device. As such, these estimations will be complicated and imprecise. Still there are no reasonable alternatives to the user providing these values, they are necessary for the shared hardware use, and there exists no reasonable way for the testbed to calculate them. As such the estimations are needed and any potential problems can be solved with error margins.

To calculate if the experiment would be executable on the testbed, the requirements need to be compared with the current infrastructure of the testbed. This information is stored in the **network graph**, the digital representation of the physical network. The network graph needs to be provided and updated by the administrators to reflect the current situation. It is a graph with all the nodes of the network both the end devices and the network nodes. The vertices between the nodes represent the network links and store information about the ports connecting the devices and the maximum bandwidth of the link.

Eventually the testbed could detect these changes automatically, but it was judged to be an unnecessary complexity. Especially because physical changes should happen rather rarely, since they require the cancellation and replanning of all scheduled experiments to ensure the planned routes were not impacted.

The core task of the scheduler is the scheduling decision, calculating if the testbed can accommodate an experiment by comparing requirements and available resources. A subproblem of this task is finding a timeslot that can accommodate the experiment. This can be done through repeating the scheduling decision with the subset of resources available at the specific time. Both these tasks are related to, calculating the schedule. Additionally, the scheduler must keep the schedule by instructing the Installers, when to start and stop experiments.

Finally, the scheduler handles a third decision process that has nothing to do with the original scheduling problem, but also relies on the network graph and is therefore assigned to the scheduler as the component responsible for the graph. The scheduling algorithm produces a list of devices and links that need to be reserved for an experiment. But the testbed requires a list of instructions on what needs to happen to achieve this described state. As such the scheduler must build the instruction list from the scheduled experiment for the installers before starting the experiment.

Scheduling Algorithm

The core problem of the scheduling task, is figuring out if an experiment can run on the available hardware. While the available hardware may change through being reserved by other experiments, this comparison forms the base of the algorithm. The algorithm is complicated through two problems. First users request end devices and a required bandwidth between the devices. On the other hand the scheduler has knowledge of the network layout. It is therefore part of the scheduling to find the connecting network nodes and links, that the experiment will then use. This is a graph problem. The algorithm needs to find a path connecting all the requested devices with the required bandwidth. Only if such a path exists the experiment can be scheduled. Optimally, the path should have the least amount of links possible to keep these resources available for other experiments.

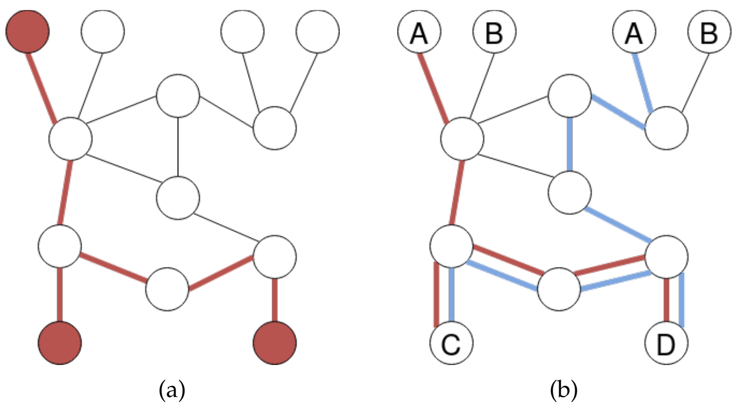


Figure 3.2.: Minimal Spanning Trees

Secondly, the class based selection system further complicates the process. It adds an additional variable that the scheduler must assign during the scheduling decision. While one specific device may not be connectable to the other requested devices, a device of the same class could then be substituted. With the network and class variable the scheduler has a lot of freedom, when it selects a path and every change to the available resources could have wide-reaching consequences for schedulability of the experiments. For example on the full network graph an experiment may be planned on a specific path. But one of the links on the path is reserved by another experiment at the time. This does not mean the experiment can not be scheduled a different, longer path may be still available or different devices of the requested classes may be still available. This means that to find the optimal resource usage the scheduler can not pre-compute the path once and then apply to the changing available resources. The complete planning step must be rerun every time.

The first step of the algorithm is comparing the requested devices. If one of the devices or device classes is not available, there is no reason to do the complicated network comparison. If the hardware generally exists the scheduler needs to find the subgraph connecting the selected hardware. Such a task is called the Steiner Tree Problem in Graphs. It is characterized as a minimal spanning tree over S with $S \subseteq V$ for the weighted graph $G = (V, E)$ [HR92]. Figure 3.2 shows what a potential network graph could look like. The leaf nodes would represent the devices bookable by the users, while the other nodes are the network devices like routers and switches. In the example the experiment requests three devices (marked red) and while there are multiple minimal spanning trees of the same length, one potential solution is the red path. Bandwidth can be considered in the calculation by first removing all links that do not have the necessary capacity and running the algorithm on the reduced graph.

While this helps in calculating the ambiguity of network connections it does not solve the class based selection problem. Figure 3.2 shows two minimal spanning trees both connecting an A, a C and a D, but of differing length. Therefore, to find the optimal spanning tree all possible combinations of nodes need to be calculated and then the smallest can be returned. All possible combinations can be described as $\prod_{A \in X} |A|$ with X being the set of all the requested deviceclasses.

Since there exists a way to find the optimal solution to the Steiner Tree Problem and the scheduling algorithm can be reduced to repeated application of the same problem, there also exists an optimal way to find the best subgraph connecting the requested devices.

Alternatively, the testbed could take a more greedy approach by combining both the network and class selection problem into a single algorithm. Shown in 3.3 (a) the greedy algorithm is searching outward from each member of the class with the fewest members. In the example, this could have been either C or D, with D being randomly chosen. In the first iteration, the algorithm checks the node with number one, continuing to search more nodes every step until, in the fourth iteration, it finds C. The algorithm then connects C and D and adjusts the distance of the already found nodes to the new path. This is shown in figure 3.3 (b). The nodes on the path have zero distance, and the other already-found nodes get reduced to distance one. Now the algorithm can repeat, searching outward until all requested nodes are found. In the example, it would find the A in the upper-left corner and achieve the same result as the optimal algorithm shown in figure 3.2. If the algorithm assigns a distance to all nodes without finding all the requested nodes, scheduling is not possible. This can happen if links are removed because they do not have the needed capacity, splitting the network into partitions.

As figure 3.3 (c) shows the greedy algorithm will not always find the optimal solution. By changing the B in the upper right corner to a C, the shorter blue path is created, but the greedy algorithm will never find it, because it terminates before reaching the relevant nodes.

While this algorithm does not produce optimal solutions it combines the network and class selection search into one step. Eventually pruning techniques designed for tree search can be utilized to further reduce the amount of search operations by combining and dropping overlapping search trees.

Timeslot selection

After judging if an experiment can run at all using the full network graph, the scheduler needs to find a timeslot in which the requested resources are actually available and

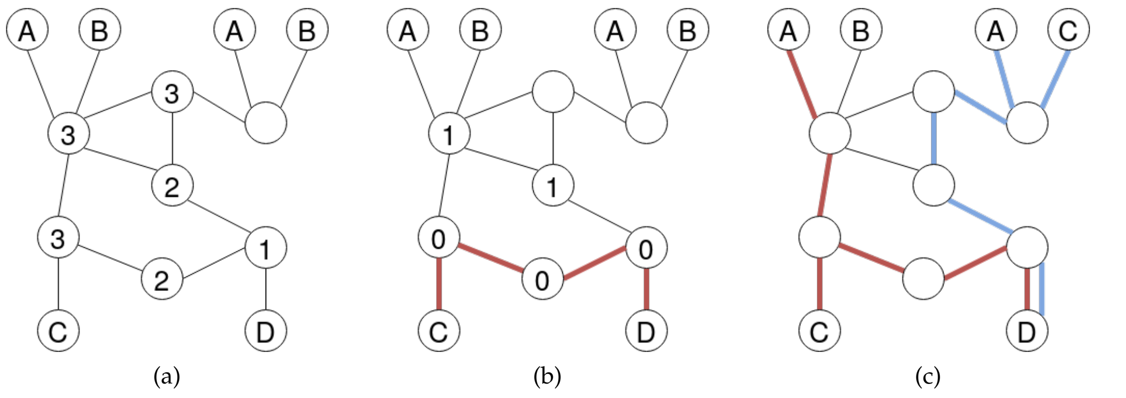


Figure 3.3.: Greedily searching for a subgraph

not already booked by other experiments. This is simply a variant on the general scheduling algorithm. By subtracting the already scheduled experiments from the full network graph, the system can calculate the network condition at a specific time. Shown in figure 3.4 is the example network with added bandwidth capacities in MBit/s. Subtracted from the graph in (a) is an experiment connecting the two greyed out nodes with a 500 MBit/s bandwidth. A second experiment requesting more than 500 MBit/s bandwidth would then utilize the graph in (b) for its scheduling decision. As the subtraction has fragmented the network some routes that have been possible before are no longer possible and the experiment could potentially no longer be scheduled.

The testbed always tries to find the next possible fitting timeslot for an experiment. Therefore, it only needs to go further in time until a fitting slot is found. While planning an arbitrary amount into the future may not be reasonable, and a cutoff point for max search time may be necessary, in the worst case the testbed could always append the experiment at the end of the schedule. As such an experiment judged “schedulable” in the first run on the full network graph, should always find a fitting timeslot.

The scheduler only needs to reevaluate the schedulability of the experiment and changes in the network graph, meaning where already scheduled experiments start and end. Most of the time the reevaluation will not require the complicated network scheduling algorithm, but a simple list comparison if the requested devices are currently available.

Finally, experiments are not points in time, but may last several hours. Therefore, the algorithm has to consider all possible colliding experiments when scheduling a new one. This could be done by simply removing all the colliding experiments from the network graph, while this would work for the end devices, where every device used by a colliding experiment is unavailable for considerations for the planned experiment. This

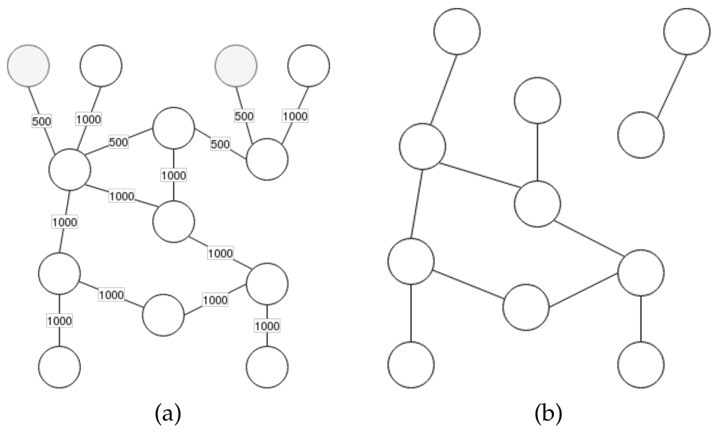


Figure 3.4.: Subtracting scheduled experiments from the network graph

would lead to wasted resources for the network links. For example, if one experiment ends then and another experiment starts during the planned timeslot both using the same network link, simply reducing the capacity by both would limit the available bandwidth more than the actual use in that timeslot. Instead, the algorithm needs to calculate the maximal reserved bandwidth by going through the colliding experiments chronologically and calculating what resources are actually used during the entire timeslot. While this leads to rather expansive preprocessing steps, that the algorithm needs to take, it is vastly preferable to run the search algorithm multiple times, as that would be vastly more computationally intensive.

When the algorithm returns the found timeslot the user is asked to confirm the booking. After this the used end devices, used network nodes and links are reserved for the experiment. The algorithm does not replan existing experiments, since these reservations are communicated to the users so that they can use device specific information in their experiments.

Instruction list

Finally, when an experiment reaches its scheduled starting point, the scheduler must instruct the installers on how to reach the state described in the experiment. It does so by sending a list of instructions to the installers. As shown in table 3.1 there are five basic instructions. From these `INSTALL` and `LINK` are mostly relevant to the installation process. `STOP` and `UNLINK` are used at the end of an experiment to return to the base state, awaiting new installations. The scheduler only includes the minimal amount of information necessary in the instructions. The scheduler expects the installers to collect the information it needs for the operation itself. As such the scheduler only includes the command, the effected device and an identifier for the experiment in the instructions send to the installer.

Table 3.1.: Possible Commands

Name	Devices	Function
`START`	end devices	Turns the device on
`INSTALL`	end devices	Installs the operating system on the device
`STOP`	end devices	Shuts down the device
`LINK`	network device	Creates a VLAN between the ports
`UNLINK`	network device	Removes the VLAN between the ports

3.3.2. Installer

Unlike the monolithic scheduler the installer is not a single component. Its goal is to outsource and parallelize the time and resource intensive installation process. The installer is not a decision-making instance of the testbed, all decisions are made by the scheduler and then the installer is merely instructed to act accordingly.

Additionally, it also adds an abstraction layer between the scheduler and the physical infrastructure. Because of the separation between scheduler and installer, the scheduler works only on internal data structures and therefore the scheduling algorithm can easily be exchanged, if different needs arise from the setup conditions. Also, the scheduler does not have to be physically part of the testbed, because it only interacts with the data from the network graph and not the network itself. Only the installer as part of the main components needs to be physically part of the testbed network.

The installation process is device specific, while there are often similar parts in the process, the general assumption is that every device comes with its own requirements and capabilities. This also follows out of the Commodity Hardware idea, these devices do not adhere to a common standard and the testbed should not force them to implement such a standard, but capabilities that exist should also be utilized. Somebody implementing the testbed for themselves using hardware that for example has a remote management interface, will want the testbed to use this function to simplify the initial setup process.

As such the testbed itself does not have any understanding on what is required to interact with the hardware. Instead, the administrator has to provide the testbed with custom commands scripts for every type of device. The script is then dynamically loaded by the installer based on the device identifier that needs to be interacted with according to the instruction list. The scripts have an extremely broad scope. They mainly instruct and configure the appropriate services running on the installer. This could for example be a DHCP server or a TFTP server. They can send messages to other devices. This could be http-requests to transfer information to other components of the installation process, for example a local cluster manager, or magic packages that are required by the wake-up functionality of some devices.

To support the installer and the command scripts the testbed provides information look up capabilities via the resolver. The resolver is one of the minor components and is tightly coupled with the database. The script may require to know the MAC- and IP-addresses of the devices in order to send messages, which are provided by the resolver. Additionally, the resolver gives the installers access to the file repository in which the operating images, uploaded by the user are stored.

Heterogeneous Hardware

Heterogeneity as part of the commodity hardware requirement is the most distinct characteristic of the Edge-Testbed. As interaction with the devices is a significant part of the operation of the testbed, there is an inherent conflict between the static nature of the system and the dynamic demands of the devices. And while there may be overlap between the processes, no single part of the process could or should be relied on.

As discussed instead the testbed uses dynamically loaded command files to interact with the devices. As shown in table 3.1 there are currently five commands, that are basically the interaction points for the installers. According to the command and the related device the installer will load the correct file to execute. Furthermore, the commands are differentiated between end and network devices. The two groups serve different purposes for the testbed and as such they are interacted with differently.

These command scripts are written by the administrators and are required to be provided for every deviceclass. Probably there will be minor differences in what data and format needs to be sent to the devices to control them. Additionally, the scripts will probably require very specific data for the interactions. For example, the remote management interface of the Dell OptiPlex requires a username and password. This information is both device specific and only relevant for this device class. The scripts can therefore pull information about the devices as required via the resolver. This way the installer does not have to have any understanding of the devices and can outsource all these processes to the command script.

While this characterization may suggest a decoupling between command script and devices, this is not actually the case. The installer loads the scripts dynamically, but the scripts heavily rely on the installer and the services that run on the same device. For example the DHCP service is not part of the installer as a program. But it is a service running on the same device, that the installer needs to configure in order to interact with the devices. As all the device interactions is handled by the command scripts, it is the script that needs to correctly configure the service, but to do so the script requires deep knowledge of the system running on the installer. If the configuration file in the DHCP example are simply moved to a different folder, the command script would no longer find it and fail to run. Resulting in, potentially minor changes to the installer device having significant impact on all the command scripts. Changing to a different DHCP server would require adjusting every single command script accordingly. This issue is further stressed with multiple installers as each instance needs to run the same services and the services have to be installed in the same way.

This issue can be solved with an additional abstraction layer on top of the system services. The abstraction layer could then provide a unified standard across the devices and allow for change on the underlying layer without affecting the command scripts.

Handling multiple Installers

As mentioned the installer does not have to be a singular instance, this may be advantageous in some case to run multiple install processes at the same time. Multiple installers may also be necessary in some cases. Some devices may require different connections or capabilities from the installers. For example to start the Intel NUCs a Raspberry Pi is manually pulling the power pin on the device's motherboard. In that case only a specific installer could be used for the devices.

Be it out of necessity or simply to facilitate faster installs, the testbed needs to facilitate the possibility of using multiple installers. To allow for multiple installers means that tasks have to be distributed between devices. There are different ways such distribution could be done by the testbed. The first would be to have one main installer that then further distributes the tasks as applicable. In this case the scheduler would send the instruction list to the main installer, the installer handles most of the processes that can be done quickly and then outsources only the time intensive parts like loading and installing the operating system images to the other installers. This could either be done with one main installer for the entire testbed or the scheduler deciding on the main installer for each experiment.

This structure has the main advantage that it outsources only the smallest part necessary, but the general control stays with the main installers. This is particularly useful for central per network services like DHCP. As multiple DHCP servers would lead to conflicts a single main installer could host the DHCP server and then refer to the other installers in the DHCP responses.

Alternatively could the scheduler assign individual tasks to different installers during the creation of the instruction list. In this system there are multiple equal installers that could all be valid for a task and the scheduler assigns the task semi-randomly for example in a round-robin-scheme. This leads to multiple equal installers instead of the more hierarchical infrastructure than in the first approach. Devices with specific requirements could be accommodated by only having their installer in the list of possible installers and central services like DHCP could be hosted by one of the installers, but accessed via an interface reachable from all other devices.

Both approaches may be reasonable depending on the scale and complexity of the testbed. In a smaller testbed the first approach may be perfectly fine, but on a larger system with multiple physically distant devices it could become no longer possible. As this mostly relies on how the scheduler is configured and the concrete implementation of the command scripts, it is up to the implementor to choose either approach or any reasonable combination. Even with the scheduler choosing the installer, a main installer could be implemented by only configuring one device and then sending further instructions from this installer to other devices using for example web-requests.

3.3.3. Resolver & Identifiers

One of the problems when working with device installations and interactions are device identifiers. Most identifiers are designed to be machine- and not human-readable. As such, differentiating between devices on the basis of, for example, their MAC-address is rather complicated for humans. The testbed solves this by using an internal identifier scheme based on URIs [BFM05]. The hierarchical nature of the URIs makes it easy to identify the device for both users and administrators. For example, the URI `tb://devices/dell/optiplex5000/702/1/1` gives users the type of device and administrators could see that this is the concrete device in room 702 on server rack 1 position 1. Human-readable identifiers make talking about the devices and quickly grasping the device information far easier.

As both the experiments and the installers still require the actual identifiers used in the related protocols, there needs to be a connection between the URI and the hardware information. The resolver provides this connection. Components can send requests to the resolver and the resolver then returns the information stored in the database for the related identifier.

For devices, it returns device class and hardware information, for device classes it returns the related devices and the command scripts and for experiments it returns the map between devices and OS images and the network configurations. What is included in the response relies on the requested identifier, as such future expansion of the testbed could utilize their own response formats.

3.3.4. Frontend

The Frontend encompasses everything that users and administrators interact with. It is therefore not a singular component but the accumulation of all user interfacing components. For the users the frontend is mainly related to the scheduling step of the experiment lifecycle. It collects the device, bandwidth and length requirements, handles the booking confirmation and finally provides an interface for uploading the operating system interface. Additionally, it provides information about the testbed. One of the reasons the testbed does not replan experiments is that users should be able to rely on hardware specific information. This information would also be provided through the frontend. Furthermore, the testbed needs to provide a list of available devices and deviceclasses. Especially since the testbed uses an internal identifier system, there needs to be a link between a device description and the identifier.

On the more general user interactions the frontend should provide them with information about scheduled and past experiments, as well as uploaded data or data collected from the experiment via the datasink.

The frontend is also the contact point for administrators, when configuring the other testbed components. Mainly these are changing the network graph to reflect physical changes to the network and uploading and linking the command scripts for the installer with the devices. Administrators should additionally have direct access to the testbed infrastructure by passing the frontend to fix potential issues from the devices like failed or partial installations.

The separation between frontend and the other components should make it easier to replace the frontend with whatever user interface the implementor prefers.

Authorization, Accountability & other Security Considerations

The testbed is not designed with security in mind. In order to facilitate the most experiments, users are given wide ranging powers over what they can do with the devices. The restrictions are minimal therefore the potential for abuse is significant. Any restriction that would limit abuse, could also stifle legitimate experiments. A malicious experiment can therefore both attack the functionality of the testbed and attack targets in the broader internet.

As the internal components are so vulnerable, security concerns need to be solved at the entrances to the system. The frontend as the user facing component is therefore responsible for authorizing the users before passing along their communication to the rest of the network. Since users are inherently trusted, outside vetting of the user is required before giving them access to the system. This means the testbed is not designed for use of the general public, instead the target is a user base that can be safely administered by a single person validating each user.

With authorization handled through external sources, it becomes important to control what users are doing with the testbed. Users have to be hold accountable for their actions to identify and correct both accidental and malicious misbehavior. As all user communication passes through the frontend, it can track the users actions both in regard to what devices were booked, but also all other communication going into or out of the testbed. This allows administrators to understand how user interactions influence the testbed.

The other security goals are only tangentially related to the functionality of the testbed. Experiments are not inherently confidential. And eventual confidential information as part of an experiment are difficult to protect by the testbed. Since the testbed has no understanding on what is potentially confidential and against what threads it would need to protect,the only protection the testbed can provide is encrypting the data transfer between users and the frontend. However, as experiments can communicate with the internet, there still exist ways for unencrypted data to leave the testbed and restricting such datastreams would once again put unnecessary limits on the testbed.

Availability may also be a potential issue, when one user books all the available resources or for an excessive amount of time. But any restriction towards booked time or resources would be entirely arbitrary and if the user acts maliciously, access to the testbed could simply be revoked by an administrator. Finally, integrity could be relevant for the file transfers that need to happen between the different components, but as these can rely on established protocols it is not relevant for the design of the testbed itself.

3.3.5. Database & File Repository

Database and file repository are the storage parts of the testbed. They store the information for all the other components of the system. Most of the information is stored in the database. For the frontend that would be user information like password, e-mail address and access tokens. For the scheduler and resolver it stores the hardware information like addresses, device types / names and the related hardware classes. Finally, the database stores the experiments with the related resources and scheduled starting and end times.

The file repository is an addition to the database for dealing with data blobs. The testbed has to handle a lot of these for the operating system images and is therefore separated from the rest of the database. The separation should allow for replication of the file repository, as files should remain unchanging, the file repository shares similar characteristics to an append-o,ly database. After retrieving the correct file identifier from the database, an installer could then load the image file from any of the file repositories.

3.3.6. Optional Components

The experiment controller and the datasink are optional components of the testbed. They provide additional services that may be useful to some experiments, but are not required to be used. The users are free to solve the issues related to the components on their own. The two functions, namely sending pre-scheduled messages to the experiment via the controller and collecting data from the experiment via the datasink, are probably part of most of the experiments. It is difficult to even consider an experiment, that does not collect data in some way. Therefore, it makes sense for the testbed to provide this functionality directly, so that multiple experiments may share the component, and they do not have to set up their own systems.

3.4. Considerations on the Bandwidth Estimation

As discussed in section 3.3.1, estimating the time and bandwidth requirements, while necessary, can be rather complicated. For estimating the required time, the user needs to have an understanding of, how their experiment will handle. For example, in a file transfer protocol, a researcher may want to measure how long a transfer will take, but to do so, they would first need to estimate the result to schedule the experiment. Furthermore, results could be lost, if the experiment is forced to terminate before collection. For the time requirement, this can be prevented through overestimation. While it leads to some resource waste, the potential loss of results is so much more significant that doubling every estimate is still viable.

While estimating the needed time can be solved through overestimation, requesting too much bandwidth could make an experiment no longer schedule-able and would eliminate the testbed's ability to schedule multiple experiments concurrently. For example, most of the links to the devices are capped at 1GBit/s. If a user now decides to select a higher bandwidth because they plan to use the full 1GBit/s and include a safety buffer, the selected devices would not be connectable. Unlike time, where everything else will just be pushed further into the future, the bandwidth is hard capped, and exceeding these caps means that no experiment is possible.

With that said it is important to consider how bandwidth is implemented by the testbed. It is **not** that every device part of the experiment is connected to every other device of this experiment with an exclusive link of the requested bandwidth. This follows directly out of the "Commodity Hardware" requirement and the layout of the physical infrastructure. As already mentioned in relation to figure 3.2 devices are leaf nodes on the network graph. They are only connected with a single link to all other devices. This is part of the "Commodity Hardware", where the testbed should not depend on devices having multiple network interfaces to connect to.

The network created by the testbed through the VLAN should not be understood as an n-n-connection, but rather as a star topology, where every leaf has the requested bandwidth to the inner node.

The testbed is an abstraction of the physical network. The inner node in the start could represent a multitude of routers and switches, that the user must no longer have knowledge of. But the physical reality needs to be considered by the user in order to place realistic requirements on the testbed as part of their experiments.

3.5. Separation between Scheduling & Execution

The two main purposes of a testbed are to make experiment set up easier and allow for reproduction of the experiments. Most of the components of the testbed focus on the first point. They either are relevant for the installation functionality or provide other supporting functions to the experiments. On the other hand reproduction was rarely considered in the design of the components.

The main approach the testbed has towards reproduction is an automatic documentation of the selected hardware and a replanning functionality based on this documentation. For this there is a conceptual separation between the end of the scheduling step and the beginning of installing. When constructing the instruction list the scheduler is also creating the experiment descriptor. The descriptor includes the original request as well as the actual devices chosen for the request. It includes the bandwidth, length and the identifiers for the operating system images mapped to the devices. This information can then be downloaded by the user to give a clear overview over the system. Additionally, the user can create a new experiment from the descriptor either an exact replica or choosing which parts need to be kept and which may be substituted by the scheduler.

With this simple record keeping measure it is much simpler for the users to keep track of their used resources and therefore accurately describing the experiment should also become easier.

4. Implementation

The design defines basic structures relevant for functionality of the testbed. The implementation further refines these concepts into a more useable and more sensible architectural layout appropriate to the scale of the current application environment. The goal is to develop a working prototype for the testbeds functionalities with the available resources, while adhering to the overall requirements of the testbed.

4.1. Physical Infrastructure

For testing and evaluation the testbed is implemented with four devices nodes connected to one switch. The testbed software components are all run from a single additional node, this device is called the testbed controller. Additionally, two nodes require further hardware in order to start and stop the devices.

The testbed nodes are two Dell OptiPlex Micro 5000 and two Intel NUC 11. The two NUC devices have no remote management capabilities. To start and stop the devices a Raspberry Pi is used to manipulate the power pins, which are pulled to the front for the NUC devices. The testbed therefore sends a POST-request to the Raspberry Pi, which runs a minimal Python Flask[1] server to close a relay connecting the power pins. This can lead to problems since the controller does not know the current power state of the device. Also in order for the testbed to intervene in the start process and overwrite an existing operating image, the device needs to initially request the DHCP server for installation images via Preboot Execution Environment (PXE). This is a time intensive process in which the device waits for a response before switching to the next boot device in the order.

The OptiPlex on the other hand implements the Intel Active Management System[2]. With that the testbed can send magic packages to the devices to start and stop the devices, as well as change the boot order. Because of these differences the OptiPlex devices are easier to handle for the testbed and restarts are a lot faster since they only need to check the DHCP server for PXE during the installation process.

[1] `https://flask.palletsprojects.com/en/3.0.x/`

[2] `https://www.intel.com/content/www/us/en/architecture-and-technology/vpro/active-management-technology/overview.html`

The devices are connected by an Aruba InstantOn 1920 network switch. The switch can manage multiple separate VLANs, supports Power over Ethernet and can be configured using the Simple Network Management Protocol (SNMP). Via the management system the testbed can instruct the switch to move ports to different VLANs. These connections can either be assigned exclusively to one VLAN, which is useful for end devices, or they can be shared between multiple VLANs via the VLAN Trunk Protocol, which is useful for connections between switches.

The testbed controller is an additional Dell OptiPlex Micro 7040. The controller uses two separate network cards to connect to the outside internet, as well as the internal testbed network. In that regard the controller acts as a router passing along internet packages for the wider internet from and to the testbed. It additionally manages the internal network using DHCP. As a router the controller is added to every VLAN, so that the experiment nodes can connect to the internet and be reachable via ssh. The controller runs on Debian 11, the routing functionality is provided via *iptables*[3] rules and for DHCP it uses Kea DHCP[4]. Kea is advantageous because the configuration is written in JSON, therefore easily machine-readable/editable, and the REST interface allows for changing the configuration without restarting the service.

4.2. Testbed components

The testbed is designed around the idea of a multitude of interacting service components. It is not a clean microservice architecture, as the applications mostly provided through the frontend are not individual components, but the general concepts are used to separate the concerns and allow for scalability of the components.

The implementation however uses a frontend-backend approach. Figure X shows this layout. The users and administrators interact with the testbed using a website, that relies on REST-backend for interaction with the database. Additionally, the backend application also includes the scheduler, resolver and datasink functionality. Each of the functionalities uses REST-endpoints to provide their services, either to the website, the installer or the physical infrastructure directly.

This change in infrastructure is done to simplify the application landscape. Instead of five different REST-servers each providing a small interface a single server can provide the functionality for all the applications. The website on the other hand is separated out, as it relies on a different technology stack than the REST-servers.

In addition to the front- and backend application the installer is its own application. Unlike with the other components the separation between scheduler and installer is

[3]`https://www.netfilter.org/projects/iptables/index.html`
[4]`https://www.isc.org/kea/`

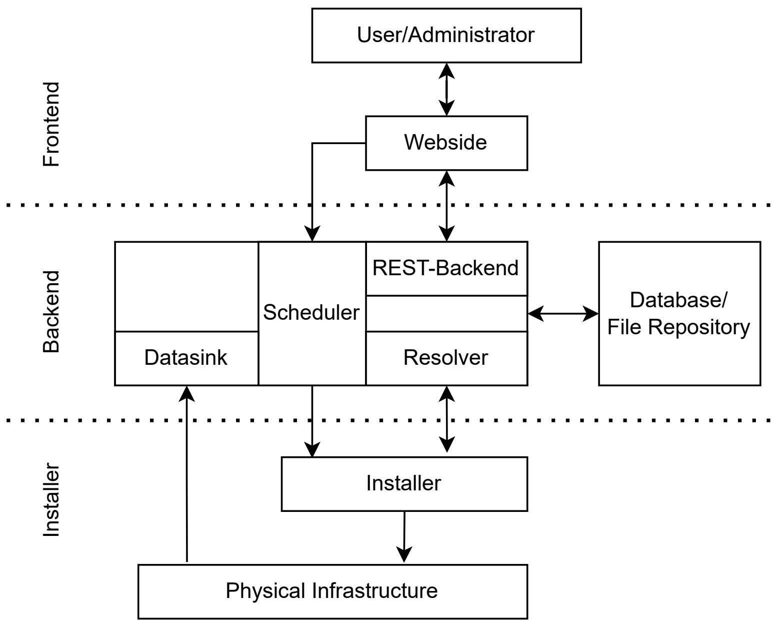


Figure 4.1.: Implementation

a core design decision of the testbed. Especially the limited information provided to the installer initially and having to then request the information from the resolver, is a core characteristic of the system and can have significant impact on the performance of the installation process. On the other hand the fact that resolver and scheduler are provided through the same application should have little impact on the overall performance as none of them is communicating with each other, and they are also not working in parallel or even on the same problem.

4.2.1. Website

The website is the point of contact for users and administrators to interact with the testbed system. The server uses Spring[5] and specifically SpringBoot to provide the website. Spring and the Java ecosystem was chosen as they provide many useful supporting libraries and the Controller-View architecture allows to easily supply dynamic webpages to the users. One important library is Spring Security that allows users and administrators to be authorized. The administrator and user functionalities can be easily separated based on URL via configuration files. Additionally, the session manger helps automatically in handling identifying users across sessions and timing

[5] `https://spring.io/`

out sessions no longer in use.

At its core the website is a presentation layer on top of the REST-backend. Information is requested from the backend and then presented to the user. For users the main two tasks are file handling and presenting information about the available devices to the users. Users need to select the devices during booking of the experiment using the internal URIs. As such, the website needs to provide the URIs of the devices, a description of the component and the hardware addresses associated with the device. For the second task the user testbed interactions are based around files. Files have the advantage that the user can easily preserve experiment data by downloading the generated files. These files are experiment descriptors with information about the selected hardware, both to recreate the experiment with the identical hardware and to resend the same request via the class based device selection system. Descriptors are provided in a JSON format with the requested devices and classes, the actually selected devices, the requested bandwidth and requested length. Also optionally they include the URIs and hashes of the operating system images and the devices they are linked to. This part is optional since the information can be provided by the user at any point between booking and experiment start, but the descriptor should be available imminently after booking. The other type of files are the operating images provided by the users. These can be associated with multiple experiments allowing reuse of the provided software. The uploaded files are also automatically associated with URIs. These URIs are then utilized to refer to the file when referencing them for the experiments.

Finally, the website provides an interface for booking experiments to the users. For this the user must select the devices either based on individual or class based URIs. Additionally, the user needs to provide the requested bandwidth and experiment length. After the booking has been scheduled by the scheduler and confirmed by the user, the user needs to link the selected devices with the operating images. For this the users can access and view the booked experiment’s data and link the URIs for the experiment and the uploaded files.

For administrators the website allows interacting with the testbed configurations. Administrators need to be able to manage users and hardware. User administration is mainly about creating and removing users. As discussed during the design with experiments being very powerful, it becomes necessary to trust in the users not to abuse the power they are given. For the implementation this is solved by administrators registering the users in the system and removing the access for users that no longer require it. Users are identified based on an email address and a short hand identifier. The short hand is used as part of the URIs associated with the users. This is part of the hierarchical system of URIs, as data belonging to individual users can be easily identified based on the URIs.

For hardware management administrators need to be able to inform the testbed about the state of the physical infrastructure. This is done via an uploaded JSON file. The JSON includes a list of all the available hardware and a list of the connections between the devices. For each device the list includes identifier, class and further hardware information like IP- and MAC-address. Then in the second part each connection is given from the view point of the switches with ports and bandwidth. So for example a switch has the connections port 1 with device X and port 2 with switch 2, and the other switch could then have port 5 with switch 1 and port 10 with device Y. Each connection includes the maximum bandwidth of the connection. This information is then transferred to the scheduler to create the network graph and the devices with their information are stored in the database. Secondly administrators can upload the command scripts to the testbed associate the scripts with a specific device class and a specific command word. Adding a new device is therefore done by uploading the changed network file and then adding the command scripts for each interaction type.

4.2.2. Backend

The Backend is a second Spring-based web server. Instead of providing webpages, it uses the Spring capabilities to provide REST endpoints to interact with the other components. As discussed for figure 4.1 the backend is a collection of services that make up the testbed functionalities. The backend includes scheduler, website backend, datasink and resolver. Each service could be implemented as its own REST service, but as there is little interaction directly between the functions, combining them into a single application reduces the overall complexity of the application landscape. The server uses a MariaDB Server for persistent datastorage, which integrates well with the overall Spring landscape using JPA to interact with the database.

The first sub-application is the backend for the website service. All functionality supported by the website is actually implemented through this part of the backend. The website passes authorized requests along to the backend where they can be fulfilled via the connected database. This separation allows for a clean front- backend design. Data presentation is handled by the frontend and data collection or request interpretation are handled by the backend. Clean separation makes the frontend more exchangeable, which is especially useful since this implementation only provides a very limited HTML + JS website, while modern web frameworks could support more modern design approaches for website development. One of the major components in handling the website is handling of the file transfers between user and system. Files are not stored in the database. The system stores the actual file in the file system, while keeping a reference to the position in the database. Files can then be retrieved by other components of the testbed through the file repository service, which loads the file based on the identifier.

Another interaction point for the website are the REST-endpoints for the scheduler. The scheduler utilizes the greedy algorithm outlined in section 3.3.1. The algorithm uses JGraphT[6] to represent and traverse the network graph. JGraphT is a java library for handling graph structures and in the case of the network graph handling an undirected, unweighted graph. The adaptable nodes and edges allow for storing all the relevant information like device names and connecting ports directly in the graph, saving unnecessary database lookups.

After reducing the graph according to the capacity requirements and removing the devices currently booked, the algorithm first connects all the specified nodes together and then searches outward form the created path to find the nodes on class selection. To connect the initially fragmented network the algorithm uses the shortest path search build into JGraphT to connect each node with the already selected network. Only in the second step once all preselected devices are connected to the path, can the greedy algorithm be used to find the rest of the selected devices.

That means in step 0 it checks all network devices already on the path if they have the requested class of devices available. In the next step it then checks the network devices one connection removed, going outwards until all available nodes are checked. If a node of the requested class is found the shortest path search can again be used to connect it with the rest of the subnetwork.

As discussed in the design this algorithm is used initially to test if an experiment is possible at all, and then reused to find the next available timeslot by adapting the network graph to the network condition at the time.

Once the experiment is booked by the user, it is stored in the database and scheduled via the Spring task execution and scheduling API. At the scheduled time the scheduler creates the task list for the installer, if all the required information (i.e. the operating system images), have been provided. For this it first adds all the LINK instruction for the network devices, followed by the INSTALL instruction for all selected end nodes. This order is chosen since the controller is part of the experiment VLAN anyway via the trunk protocol, to allow for example ssh access, but if the VLAN config changes during the installation process of a device, it could interrupt the installation, bringing the testbed into a complicated half-state. After this the experiment is ready and running until the end of the timeslot, where the testbed once again creates a task list to UNLINK and STOP the devices. First, stopping all the devices to prevent the experiment from leaking messages into the full network. A command in the tasklist is the commandword (see table 3.1) followed by the identifier of the experiment followed by the identifier of the node. All other information can be extruded from this information and this

[6]`https://jgrapht.org/`

information is both needed and available for every single command word. Including more information like for example the identifier of the image to install for the INSTALL instruction would mean that the installer has to separate each instruction by type and then path this information along to the command script.

4.2.3. Installer

The installer is at the same time an incredibly complicated part of the the overall installation process and a very simple application.

The actual installer application that is part of the software developed for the functionality of the testbed, is a simple python flask web server with a single endpoint. Given the tasklist from the scheduler the installer downloads the appropriate command scripts, which are also written in python, and executes them. The information provided by the scheduler experiment and node identifier are passed along as command line arguments to the call of the python script.

Given the generality of the installer, the command scripts both can do a lot and have to do a lot. The script have an immense freedom in what needs to happen to interact with the devices, but they have to implement the required functions without much assistance of the testbed.

The START and STOP instructions are rather simple for the OptiPlex it is sending a single magic packet using MeshCommander[7]. For the NUC it is sending a single REST request to the Raspberry Pi managing the system. But even there is hidden complexity. First, information needs to be requested from the resolver about the devices. The mesh commander requires the IP address and for the NUCs the IP address of the Raspberry Pi is required. This leads to widely different requirements on what data is required to be stored in the backend. Second, the testbed does not provide the software that is run on the Raspberry. This has to be created by the administrator as part of the setup process for the device. It is up to the administrators to provide the interfaces that are required to interact with the devices.

The most complicated task is the INSTALL instruction. In general, it is separated into four parts. First, the installer must collect information. With the experiment identifier the operating system image can be found and downloaded via the resolver. Next, the image has to be provided via the Simple File Transfer Protocol (SFTP) or via HTTP with a separate bootloader. After this the DHCP entry for the device must be changed to include the server and path to the operating image. Finally, the installer must start the device. On start the device tries booting using PXE, this pulls the DHCP information and then tries downloading and installing the image.

[7]`https://www.meshcommander.com/meshcommander`

Similar to the START and STOP processes this also relies on additional infrastructure and services, that the administrator must develop when setting up the testbed. The used Kea DHCP server already provides a REST API for easier interactions, but unpacking and correctly placing the files for the SFTP server is dependent on the individual setup of the installation and configuration of the server itself. Therefore these function need to either be coded into the command scripts or a separate API needs to be added for the scripts to interact with. Since all these additional functions depend on the selections the administrators made in setting up the testbed, they are not part of the overall testbed software.

5. Evaluation

In evaluating the functions of the testbed there are three points of interest. First the testbed needs to be able to run experiments interference free. Traffic generated from one experiment must not impact the bandwidth of another experiment. This is one of the core requirements of a functioning testbed and further complicated through the multi-tenant system and the shared control and data channel. Secondly most of the developed components of the testbed deal with the installation of devices. As such measuring the time required for the installation process is useful for evaluating usability. Finally the testbed is also a management system. Its decision making capabilities are part of the scheduler functionality, which is designed for larger network then what is currently available as part of the prototype. Measuring how the scheduler handles growing networks give an idea on extendability of the testbed.

5.1. Interference Experiment

Non-Interference is the defining characteristic of the testbed. Without this the functionality as a testbed can not be guaranteed as all produced results of the experiments, would not be reliable. To analyze this IPerf[1] is used. The experiments are shielded from inter-experiment interference via VLANs. As such the bandwidth is compared between a base case, where no other system is using the network, two pairs of nodes trying to send data to each other and two pairs of nodes separated by VLAN trying to send data.

IPerf can be used to measure the maximal bandwidth between two devices. In UDP mode the client node sends packets to the server node as a specified bandwidth. The software records the bandwidth, jitter and lost packets of the connection. For the experiments a bandwidth of 1GBit/s is used as this is higher than the available bandwidth of the link and could therefore lead to problems, when experiments interfere with one another. IPerf can also be used as a traffic generator, sending a constant amount of packets to the server node. This is used to simulate another experiment running at the same time, also trying to take up the 1GBit/s of available bandwidth.

[1]https://iperf.fr/

Table 5.1.: Results of the bandwidth measure between two nodes

	send [MBit/s]	received[MBit/s]	jitter [ms]	drop rate
BASE CASE	787,2	783,4	0,0168	0,01%
WITHOUT VLAN	775,6	772,2	0,017	0,01%
WITH VLAN	758,2	753,6	0,0172	0,01%
INTERFERENCE	758,2	417,2	0,0216	43,80%

For the experiment, the two Intel NUC devices are used to measure the bandwidth between them. Each experiment is repeated five times, and the average is shown in table 5.1. As part of each individual run of the experiment, IPerf sends 1GBit/s for ten seconds and measures the average bandwidth, jitter and packet drop rate over the timeframe. The results are not expressive. The relevant characteristic is the difference between the sent and received bandwidth, as this shows how the network, not the circumstances on the device, impacted the result. The base case, with only a single IPerf connection for measuring, and the second case, with a second connection running interference, are barely different. This makes sense considering that the switch used is a managed switch. The switch would not broadcast the messages to all the available ports, instead using a *Source Address Table* to pass messages along. The experiment with the VLAN between the NUCs enabled also does not significantly differ from the other two experiments. Interestingly, the maximum bandwidth sent seems to be lower than the base case, but as table 5.2 shows, the base case had a wider variance than the VLAN experiment, with the lowest recorded measure for the base case being the average of the VLAN case.

The final measured case is that both data streams are directed to the same receiver. In this case, interference between the two applications is guaranteed, and this gives an example of what interference in the most extreme case would look like. The result is as expected: the received bandwidth is nearly halved, and there is a packet loss rate of nearly 50%. The jitter however has not significantly increased. Probably the internal packet queue in the switch is filled rather quickly, and all following packets are either slowed down by the length of the queue or dropped, meaning that there is no difference in transfer time between the packets, which would lead to jitter.

Table 5.2.: All results measured for the base case

	send [MBit/s]	received[MBit/s]	jitter [ms]	drop rate
BASE CASE				
1	756	753	0,024	0,01%
2	817	813	0,019	0,00%
3	785	781	0,014	0,01%
4	804	800	0,013	0,01%
5	774	770	0,014	0,01%
AVG	787,2	783,4	0,0168	0,01%
WITH VLAN				
1	757	747	0,017	0,01%
2	759	756	0,016	0,00%
3	758	755	0,013	0,01%
4	759	756	0,022	0,00%
5	758	754	0,018	0,01%
AVG	758,2	753,6	0,0172	0,01%

5.2. Installation Experiment

The installation is one of the core mechanics provided by the system components and requires the interaction of a significant amount of components. The installer needs to be instructed, the command file loaded, the associated operating image downloaded from the file repository and then the command file executed, interacting with the underlying services required to install the image. While this is missing the initialization of the VLAN, the installation is far more complex and labor-intensive in comparison and requires multiple transfers of big chunks of data. For the experiment a clean debian image was installed onto the devices. Starting point for the measuring was sending the instruction list to the installers. By excluding the scheduler from the process the starting time can be reliably measured. As stopping point the first successful SSH connection is used. The controller will try polling the address of the installed device until it gets an response. This is a very useful point to measure against, while it relies in parts on the installed image, it is the point of interest for the user of the testbed, as at this point the installed software is ready to use and the experiment may start.

The experiment is run with one and two of the Dell OptiPlex, as well as both OptiPlex and one Intel NUC and only the single NUC for comparison. Each experiment configuration is run five times to limit outliers effecting the result. The result is shown in table 5.3. It is clear that more devices do not seem to effect the result significantly.

Both the one OptiPlex and the two OptiPlex run require roughly the same time. This can probably be explained by the slow TFTP installations running in parallel and only the by comparison fast transfers between file repository and installer are run sequentially. Instead the dominating factor seems to be the device to be installed onto. With a single NUC taking roughly double the time of the OptiPlex to install. This shows that the additional steps from the separation between scheduler and installer are not effecting the overall result. But it also makes it difficult for users to estimate how long their images will need to install, as this would once again require deep knowledge of the underlying hardware.

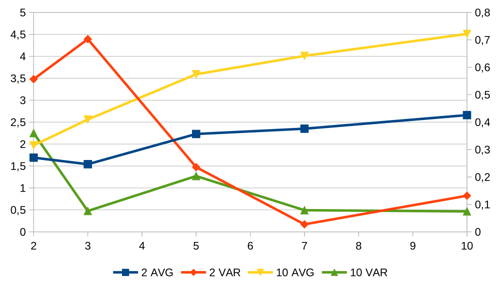
Table 5.3.: Measured time till response [in seconds]

	1 DELL	2 DELL	2 DELL + 1 NUC	1 NUC
1	231	280	481	509
2	246	217	535	505
3	248	240	495	524
4	258	210	533	494
5	267	233	502	516
AVG	250	236	509,2	509,6

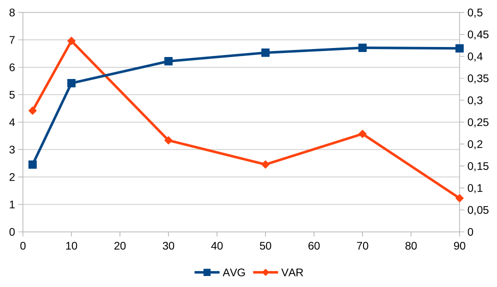
5.3. Scheduler Experiment

Allocating the available resources to the experiments is the another core task of the testbed. With the scheduler and the scheduling algorithms being a major component developed for the testbed, it is a reasonable part to be evaluated. Evaluating the scheduler with the current physical infrastructure is not possible. With just four devices attached to a single switch, the task is completely trivial. This is further amplified by the assumption, that every node is only connected to one node. Meaning that the scheduler does zero scheduling decisions on the current infrastructure. Instead it would make more sense to look at potential future applications with lots more nodes and switches. Designing such a realistic, future network is complicated. The most realistic system is probably a star topology with nodes connected to tier one switches and then these switches being connected with higher tiered ones. While such a setup would be most realistic, mimicking typical datacenter designs, it is also not interesting from a scheduling perspective.

The experiment therefore uses a randomly generated highly distributed network with each switch being only connected to two other switches. The resulting network is more similar to a ring with random shortcuts between individual elements. The end



(a) 10 total nodes



(b) 100 total nodes

Figure 5.1.: Varying amounts of selected nodes with either two or ten switches

nodes are then randomly distributed between the switches. This should lead to more complicated planning situations for the system.

There are three tasks associated with the scheduler: Finding a valid configuration, finding a valid configuration on a reduced set of the available network, and finding a timeslot. As discussed in section 3.3.1 all these problems can be reduced to the repeated application of "finding a valid configuration" algorithm. Therefore, only this one gets evaluated in the experiment. To evaluate the performance of the scheduler, the time taken in milliseconds is measured for differing configurations.

There are two variables to the layout of the network: amount of switches and amount of end nodes. Additionally, the algorithm can be asked to combine any amount of nodes into a configuration. With the random characteristic of both the network and the chosen nodes, the experiment has to be repeated to limit the effects of lucky situations. Therefore, every configuration is repeated ten times and the average as well as the variance is plotted out for the interpretation.

The first test series focuses on varying the amount of nodes that are selected as part of the experiment. In figure 5.1 (a) the amount of total nodes is fixed to ten with either two or ten switches connecting the nodes, while figure 5.1 (b) connects a total of 100 nodes with 10 switches. Both graphs show the same overall trend with the amount of time in millisecond increasing towards an upper limit while the variance decreases. This is the expected behavior, since the complicated part of the algorithm is connecting the switches, adding more nodes does not lead to significant increase in time taken. Once all switches are connected, adding more nodes to the constructed sub-tree is trivial. Therefore, the time taken reaches a cap. Decreased variance between the results is also expected, when only a few nodes are selected it is far more likely that the nodes are in a lucky configuration with a short path between them. Once all nodes are selected for the experiment, all switches have to be also connected.

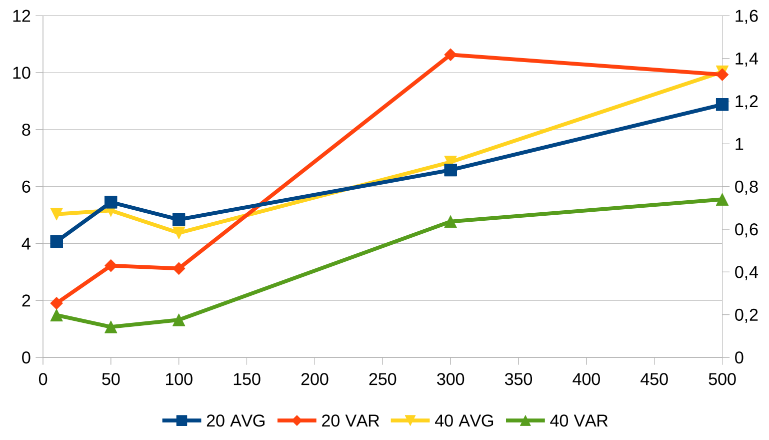


Figure 5.2.: Varying amounts of total nodes with either 20 or 40 switches

Surprising is the relation between the ten and the hundred nodes version of the experiment. Both show similar time taken for the same amount of selected nodes, the increase in nodes does not seem to have an impact on the result.

The results in figure 5.2 and 5.3 confirm these findings. In figure 5.2 the total amount of nodes gets increased, once for 20 and once for 40 switches. While the time taken only doubles the amount of nodes increases by a factor of 50. Surprising here is that both the 20 switch and the 40 switch version take nearly the same amount of time. The same result can be seen in figure 5.3, where the time taken increases only slightly between 10 and 200 switches. This is surprising since the biggest parts of the select algorithm are related to handling network devices.

At the same time the variance is rather stable. This points to the shortest path algorithm being very stable with reuse of intermediate results. Additionally, this could have something to do with the way the example network is generated. Because of the "short cuts", the shortest path algorithm will probably not consider all the nodes, when connecting the fragmented sub-network.

All this shows the scheduler can deal with probably all realistic configurations of the physical infrastructure. Also the repeated search for the timeslot should not be a significant problem as the even the highest measurements are under 11ms. Meaning that the scheduler can search through hundreds of potential timeslots before the user would even perceive the website as slow.

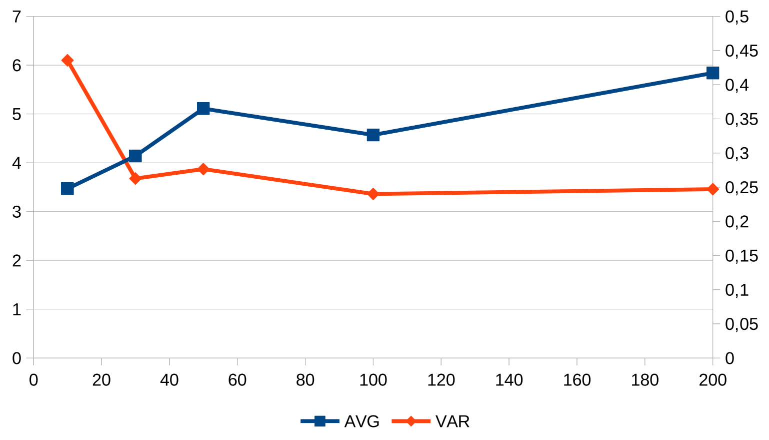


Figure 5.3.: Varying amounts of switches

6. Conclusion & Outlook

The testbed at this time only fulfills the basic functions required to function. Namely, allocation of resources, installation of devices and non-interference between experiments. As laid out in chapter 3 there are additional components not necessary to function but generally helpful to assist the user in the overall experiment cycle.

The components of the testbed are designed around adaptability. The most significant part in that regard is the command file system, allowing maximal adaptability to implement every possible technology stack needed. This adaptability comes at a cost. Mostly, that administrators must develop these interactions without any help from the testbed, and while some of the relevant steps will often overlap, every new device type needs to be considered from scratch. The administrator must judge the capabilities and limitations of the new device and develop new ways for the testbed to interact. Adaptability is also a significant strength of the testbed. Both CloudLab [RET14] and the Chameleon Testbed [Kea+20] mention in their analyses papers, that the user requirements change regularly during the operation of the testbed. For this, an edge testbed is uniquely suited. The adaptability associated with the field translates into the adaptability required by the users.

Looking at future work, additional research into interference maybe needed. Here the evaluation focuses on bandwidth, as this was the defining characteristic discussed throughout the design and implementation of the testbed. But even there, the measured bandwidth does not stay identical over multiple runs. This could be a potential problem if the full bandwidth is ever used by the running experiments. For the time being, this is solved by limiting the bandwidth to less than maximally available during scheduling. Further work into the scheduler, while scientifically interesting, especially in regard to what is a realistic setup for an expanded testbed, seems not necessary at the time. It seems unlikely that the testbed will ever reach a scale at which the scheduler could be considered a problem.

Potential for future work exists among the installers. Analyzing what processes can run in parallel and how to select the optimal installer for a specific experiment or device is another resource allocation problem that will require solving. Both the hierarchical approach with one installer further distributing tasks to devices that can solve the problem more efficiently or at all, depending on the device, and the scheduler doing the distributing, could be valid approaches. But implementing the approaches would depend on a much bigger testbed that would require the use of multiple installers.

The main task for the future will be developing more command files to add more devices to the system. The edge exists because of its devices, and therefore, an edge testbed also only makes sense if it has various devices to experiment with available.

A. Evaluation Results

A.1. Interference Experiment

Results from the Interference experiment.

	send [MBit/s]	received[MBit/s]	jitter [ms]	drop rate
BASE CASE				
1	756	753	0,024	0,01%
2	817	813	0,019	0,00%
3	785	781	0,014	0,01%
4	804	800	0,013	0,01%
5	774	770	0,014	0,01%
AVG	787,2	783,4	0,0168	0,01%
WITHOUT VLAN				
1	757	754	0,017	0,01%
2	837	833	0,021	0,01%
3	767	763	0,017	0,01%
4	757	754	0,011	0,00%
5	760	757	0,019	0,00%
AVG	775,6	772,2	0,017	0,01%
TO SAME DEVICE				
1	759	445	0,018	40,00%
2	760	410	0,029	45,00%
3	757	412	0,02	44,00%
4	754	408	0,021	45,00%
5	761	411	0,02	45,00%
AVG	758,2	417,2	0,0216	43,80%
WITH VLAN				
1	757	747	0,017	0,01%
2	759	756	0,016	0,00%
3	758	755	0,013	0,01%
4	759	756	0,022	0,00%

	send [MBit/s]	received[MBit/s]	jitter [ms]	drop rate
5	758	754	0,018	0,01%
AVG	758,2	753,6	0,0172	0,01%

A.2. Scheduler Experiment

Complete measurements of the scheduler experiments.
All the recorded results are in milliseconds.
First series with varying amounts of nodes selected as part of the experiment. One part with two switches and the second series with ten switches.

nodes select switches	10 2 2	10 3 2	10 5 2	10 7 2	10 10 2	10 2 10	10 3 10	10 5 10	10 7 10	10 10 10
1	2,2	2,3	2,3	2,1	2,2	2,5	2,5	4,3	4	4,7
2	0,8	0,9	0,9	2,5	2,5	2,2	2,3	3,3	3,9	4,8
3	2,3	2,6	2,4	2,3	2,6	2,3	2,9	4	4,2	4,7
4	2,3	2,8	2,3	2,5	2,3	2,2	3	3,7	4,7	4,4
5	0,9	0,9	2,2	2,5	3,3	2,3	2,3	3,6	3,7	4,6
6	2,2	0,9	2,6	2,2	2,4	2,1	2,7	4	3,9	4,8
7	0,8	0,9	2,2	2,2	3,1	2,2	2,4	3,3	3,8	4,3
8	0,8	2,3	2,5	2,2	3	2,2	2,2	3,5	3,9	4
9	2,3	0,9	2,5	2,5	2,7	0,9	2,8	2,7	3,9	4,6
10	2,3	0,9	2,4	2,5	2,5	0,8	2,5	3,5	4,1	4,2
AVG	1,69	1,54	2,23	2,35	2,66	1,97	2,56	3,59	4,01	4,51
VAR	0,56	0,70	0,24	0,027	0,13	0,36	0,076	0,20	0,08	0,07

Second series with varying amount of nodes selected for the experiment. This with 100 nodes available in total and ten switches connecting the nodes.

nodes select switches	100 2 10	100 10 10	100 30 10	100 50 10	100 70 10	100 90 10
1	2,4	5,3	6,8	6,6	6,8	6,5
2	2,5	5	6,5	6,3	6,1	7,1
3	2,8	4,4	5,1	6,7	7,1	6,9
4	1	4,9	6,4	6,4	6,2	7,1
5	2,7	5,8	6,4	6,6	6,8	6,7

nodes select switches	100 2 10	100 10 10	100 30 10	100 50 10	100 70 10	100 90 10
6	2,6	6,2	6	6,2	6,7	6,4
7	2,6	4,7	6	6,6	6,6	6,4
8	2,6	6	6,3	6,2	6,1	6,6
9	2,5	5,6	6,4	6,2	7,4	6,4
10	2,8	6,3	6,3	7,5	7,3	6,8
AVG	2,45	5,42	6,22	6,53	6,71	6,69
VAR	0,28	0,44	0,21	0,15	0,22	0,08

Third series with increasing amounts of total nodes. One part with 20 switches and one with 40 switches. Selecting five nodes every time.

nodes select switches	10 5 20	50 5 20	100 5 20	300 5 20	500 5 20	10 5 40	50 5 40	100 5 40	300 5 40	500 5 40
1	4,4	5	5,4	5,7	8,6	5,3	5,3	4	5,7	10,4
2	4,1	4,1	4,9	6,9	8,5	5,3	5,5	3,8	7,3	10,9
3	3,7	5	4,1	6	10,5	5,3	5,4	4	5,9	10
4	3,6	5,8	5,6	9,7	10	4,8	5,3	4,8	7,6	8,4
5	3,9	5,9	4,8	7	7,3	5,3	4,5	4,2	7,2	9,9
6	4,1	5,7	4,7	6,3	10,1	5,3	4,8	4,7	6,8	10,4
7	3,2	5	4,4	6,1	9,2	3,9	5,6	5,1	7,6	9,8
8	4,4	5,7	4,4	6,3	8,5	5,1	4,9	4,4	6,3	8,8
9	4,3	6,2	6	5,5	7	4,8	4,8	4,6	6,2	11
10	5	6,1	4,1	6,3	9,1	5,2	5,5	4,1	8	10,7
AVG	4,07	5,45	4,84	6,58	8,88	5,03	5,16	4,37	6,86	10,03
VAR	0,25	0,43	0,42	1,42	1,32	0,2	0,14	0,18	0,64	0,74

Fourth series focussed on varying amount of switches. Using to nodes and selecting half of them for the experiment.

nodes select switches	10 5 10	10 5 30	10 5 50	10 5 100	10 5 200
1	3	4,6	5	4,3	6
2	3,3	4,3	5,7	5,3	5,6
3	3,2	5	5,9	4,2	5,8

nodes select switches	10 5 10	10 5 30	10 5 50	10 5 100	10 5 200
4	4,2	4,1	4,9	4,8	5,3
5	4,5	3,7	4,4	4,8	6,6
6	3,3	4,3	5,2	4,6	6
7	2,4	3,3	5,1	4,7	5,5
8	3,8	4,3	4,2	3,6	5,9
9	2,9	4,3	5,4	4,3	6,6
10	4,1	3,5	5,3	5,1	5,1
AVG	3,47	4,14	5,11	4,57	5,84
VAR	0,44	0,26	0,28	0,24	0,25

List of Figures

List of Tables

Bibliography

[Bak16] M. Baker. "1,500 scientists lift the lid on reproducibility." In: *Nature* 533.7604 (May 2016), pp. 452–454. ISSN: 1476-4687. DOI: 10.1038/533452a.

[BFM05] T. Berners-Lee, R. T. Fielding, and L. M. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. Jan. 2005. DOI: 10.17487/RFC3986.

[Cao+20] K. Cao, Y. Liu, G. Meng, and Q. Sun. "An Overview on Edge Computing Research." In: *IEEE Access* 8 (2020), pp. 85714–85728. DOI: 10.1109/ACCESS.2020.2991734.

[Dup+19] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. "The Design and Operation of CloudLab." In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. July 2019, pp. 1–14.

[FW21] F. Fidler and J. Wilcox. "Reproducibility of Scientific Results." In: *The Stanford Encyclopedia of Philosophy*. Ed. by E. N. Zalta. Summer 2021. Metaphysics Research Lab, Stanford University, 2021.

[Hal+15] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer, and O. Koufopavlou. *Software-Defined Networking (SDN): Layers and Architecture Terminology*. RFC 7426. Jan. 2015. DOI: 10.17487/RFC7426.

[HJG20] M. Haden, B. Jaeger, and S. Gallenmüller. "I8-Testbed: Introduction." In: *Network* 61 (2020).

[HR92] F. K. Hwang and D. S. Richards. "Steiner tree problems." In: *Networks* 22.1 (1992), pp. 55–89. DOI: https://doi.org/10.1002/net.3230220105. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/net.3230220105.

[Ioa05] J. P. A. Ioannidis. "Why Most Published Research Findings Are False." In: *PLOS Medicine* 2.8 (Aug. 2005), null. DOI: 10.1371/journal.pmed.0020124.

[IT18] P. Ivie and D. Thain. "Reproducibility in Scientific Computing." In: *ACM Comput. Surv.* 51.3 (July 2018). ISSN: 0360-0300. DOI: 10.1145/3186266.

[Kea+20] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs. "Lessons Learned from the Chameleon Testbed." In: *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[LLG19] L. Liu, H. Li, and M. Gruteser. "Edge Assisted Real-Time Object Detection for Mobile Augmented Reality." In: *The 25th Annual International Conference on Mobile Computing and Networking*. MobiCom '19. Los Cabos, Mexico: Association for Computing Machinery, 2019. ISBN: 9781450361699. DOI: `10.1145/3300061.3300116`.

[LWB16] P. Liu, D. Willis, and S. Banerjee. "ParaDrop: Enabling Lightweight Multi-tenancy at the Network's Extreme Edge." In: *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. 2016, pp. 1–13. DOI: `10.1109/SEC.2016.39`.

[Ott+05] M. Ott, I. Seskar, R. Siraccusa, and M. Singh. "ORBIT testbed software architecture: supporting experiments as a service." In: *First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*. 2005, pp. 136–145. DOI: `10.1109/TRIDNT.2005.27`.

[PH12] H. Pashler and C. R. Harris. "Is the Replicability Crisis Overblown? Three Arguments Examined." In: *Perspectives on Psychological Science* 7.6 (2012). PMID: 26168109, pp. 531–536. DOI: `10.1177/1745691612463401`. eprint: `https://doi.org/10.1177/1745691612463401`.

[Ray+05] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh. "Overview of the ORBIT radio grid testbed for evaluation of next-generation wireless network protocols." In: *IEEE Wireless Communications and Networking Conference, 2005*. Vol. 3. 2005, 1664–1669 Vol. 3. DOI: `10.1109/WCNC.2005.1424763`.

[RET14] R. Ricci, E. Eide, and C. Team. "Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications." In: *; login:: the magazine of USENIX & SAGE* 39.6 (2014), pp. 36–38.