Proto^{2}Testbed: Towards an Integrated Testbed for Evaluating End-to-End Security Protocols in Satellite Constellations

Martin Ottens, Jörg Deutschmann, Kai-Steffen Hielscher and Reinhard German
Friedrich-Alexander-Universität Erlangen-Nürnberg
Computer Networks and Communication Systems
Erlangen, Germany
Email: {martin.ottens, joerg.deutschmann, kai-steffen.hielscher, reinhard.german}@fau.de

***Abstract*—There are many approaches to testbed systems in computer networks, but especially when evaluating or developing new security protocols and testing their behavior when used via satellite networks, many challenges remain that no existing system has fully addressed.**

With *Proto^{2}Testbed*, we present an approach for a new testbed system that relies on isolated virtual machines. Once a specific testbed is defined, many steps of a typical testbed lifecycle, like testbed setup and experiment coordination, are fully automated. The testbed can be used in multiple ways; one example is the integration of a satellite mega-constellation simulation, thus turning the testbed system into a complete end-to-end emulation environment.

Examples are used to demonstrate that *Proto^{2}Testbed* can be used for various experiments in its current state. The design of future experiments is outlined, which will include a testbed for extensive evaluations of end-to-end security protocols.

***Index Terms*—Testbed, Satellite communications, Security protocols, Emulation, Virtualization.**

I. Introduction

With the increasing prevalence of Internet access via mega-constellations using Low Earth Orbit (LEO) satellites, research in this area is becoming particularly relevant. Many emulation and simulation environments focus on different research fields, such as simulating physical satellite links or various routing algorithms for mega-constellations. Alongside many technical challenges, it is also becoming evident that the future development of communication protocols must consider their utilization via satellite networks. Next to existing end-to-end security protocols such as *QUIC*, *TLS*, *OpenVPN*, *IPSec* and *WireGuard*, especially Post-Quantum Cryptography (PQC) implementations of these protocols represent a particular challenge in this regard due to their more complex Key Encapsulation Mechanisms (KEMs).

Simulation models are a common tool in developing and evaluating new communication protocols. Since discrete event simulators such as *ns-3* or *OMNeT++* run in simulation time with no restrictions from real-time, they are particularly suitable for simulating complex satellite mega-constellations. However, in order for new protocols to be integrated into such a simulator, simulation models of the protocols must be implemented, which often have to deviate significantly from real implementations due to the available programming interfaces in the simulator and simplifications. For this reason, the simulation model must be maintained during development in addition to the real implementations, which requires additional effort. Likewise, results from the simulator may not reflect the real behavior of applications utilising these security protocols and, for example, access functions of the network stack of a Linux Operating System (OS).

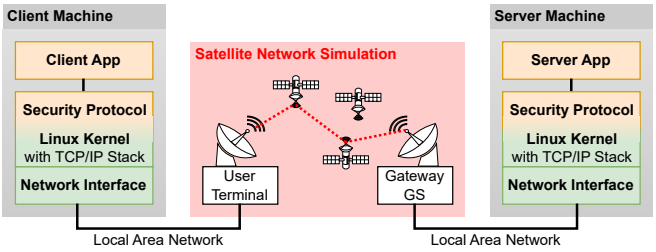


Fig. 1: Concept of how a suitable testbed with two hosts and a satellite simulator could be set up.

To achieve an optimal workflow and results when evaluating the performance of different end-applications that use end-to-end security protocols over satellite mega-constellation networks, a testbed is required in which existing, real implementations of the protocols can be easily integrated.

The paper is structured as follows: Section III compares different network testbeds with regard to the requirements presented in Section II and subsequently motivates the development of an own testbed system. In Section IV the design concept and details of the initial implementation steps of our testbed system, *Proto^{2}Testbed*, are presented. Section V shows examples of how the testbed system can be used in its current state. Finally, Section VI describes further steps that will be done to integrate *Proto^{2}Testbed* in an emulation environment, thus allowing it to become a fully integrated testbed for testing end-to-end security protocols via simulated mega-constellations. Section VII provides a conclusion.

II. Testbed Requirements

A concept of how we envision such a testbed system can be found in Figure 1. We have defined the following requirements for such a testbed system:

1. Virtual Machines and Interaction

Each end-application and the associated protocols should be operated in an isolated environment so that it appears to the end-application and protocols as if they are being operated on their own machines. For this reason, some form of virtualization is necessary. Certain protocols, such as *WireGuard* [1], are installed as a kernel module, which restricts the use of container virtualization tools such as *Docker*, especially when different testbeds are running on one workstation in parallel. For this reason, Virtual Machines (VMs) should be used in the testbed. To allow researchers to debug protocols, applications, or testbed setups, it should be possible to interact with the VMs in real-time, e.g. via a shell.

2. Different Network Topologies

It should be possible to set up different network topologies between the testbed's VMs. For example, an emulation tool simulating an environment that appears for the VMs as they are communicating via a satellite mega-constellation should be integrated. Many simulators can run in a real-time mode. In this mode, real software on the host system can interact with the simulation via virtual network interfaces [2]. The testbed system should provide the flexibility to accommodate different network paths and topologies, this ensures that the testbed system can be used in other areas of research or development. This way it should also be able to integrate real hardware or a link-emulation tool into a testbed topology.

3. Testbed Definition and Automation

Researchers and developers should be able to define testbeds easily. The configuration should be designed so that it can be generated both by users and by software. A testbed definition includes the configuration of the VMs, the network topology, the management of external emulation tools, and which experiment data is automatically collected from the testbed. With the testbed definition, the system should be able to carry out all the necessary steps for experiments without manual intervention, enabling repeated, fully automated experiments with a single testbed definition.

4. Reproducibility and Comparability

The testbed system should provide reproducible and comparable results. For example, it should be possible to compare a constant configuration of the VMs, e.g. a specific protocol version, when communicating via emulated environments with different parameters – these could be tests using different satellite constellations. It should also be possible to compare adjustments to the protocol settings or entire protocols with each other, in which case the parameters of the emulation environments remain unchanged.

III. Related Work

There are countless different approaches in the field of network testbeds, many of which pursue different objectives; they are often purpose-built and proprietary [3]. This section reviews selected existing approaches from literature. These include approaches such as ns-3 *Direct Code Execution (DCE)* [4] or *VMSimInt* [5], which allow application code and thus also real protocol implementations or entire operating systems to be run inside a simulator. In addition to various approaches for completely physical testbeds, such as *EmuLab* [6], there are also many approaches that use virtualization to provide test environments. Examples of these include *Mininet* [7] or distributed systems such as *Distrinet* [8]. Other approaches, such as *OpenBACH* [9] or *VITO* [10], focus on the provision of user-defined testbed environments, the automated execution of experiments, and the collection of result data.

A. Run Operating System or Application in a Simulator

There are multiple existing frameworks for the simulation of mega-constellations, such as *Hypatia* [11] or *ns-3-leo* [12]. These frameworks share the common feature of relying on discrete event simulators, which means they do not have to take real-time limitations into account. A simulation for a large constellation could run much longer than the simulated time. When a simulator can execute real implementations, it is sometimes also suitable as a testbed system.

With DCE, *ns-3* offers an approach for integrating real protocol implementations and applications into a simulation without major code modification or special simulation models. For this purpose, system calls are interpreted and translated into simulator events [4]. In addition to this major advantage, however, there are some drawbacks to this approach. For example, no real-time interaction with the application is possible. This also includes that application processing delays are disregarded, as an infinitely fast processor is assumed to ensure reproducibility, i.e. no simulation time elapses during the actual processing. The OS interface simulated by DCE can also cause issues with some protocol implementations, and the runtime and scalability of such complex simulations can quickly pose major challenges. Depending on the simulated environment and, in particular, the volume of traffic, such a simulation process can also run for a long time, with the simulation having to be repeated in full each time a change is made. Currently, DCE seems to be not well-maintained. Therefore, other approaches should be considered for new testbeds, especially if they need to be future-proof.

VMSimInt uses a modified version of the virtualization tool *QEMU* to integrate entire VMs into a simulator [5], eliminating the aforementioned problems with OS interfaces. However, the VMs also run in simulation time, so any interaction is impossible, and performance and scalability are questionable. With this approach, there is also no publicly available solution that is actively maintained. This is particularly problematic as new OS versions could also require future updates of the underlying, modified virtualization tool.

B. Virtualized Testbed Systems

In addition to approaches that rely entirely on simulation, there are testbeds in which real applications and protocols run isolated from each other in a container-virtualized environment, e.g. using namespaces of the Linux kernel, and are connected via a virtual network.

A prominent example is *Mininet* [7], which is focused on Software Defined Networks (SDNs) and allows the users to easily describe testbed setups, hosts, and the topology using scripts [3]. As all virtualized hosts share the host system's kernel, *Mininet* cannot be used to adequately test certain security protocols, e.g. *WireGuard*. *Mininet* also does not include functions for automated experiment orchestration, and some modifications are necessary to integrate an emulation tool into the network topology of a testbed. Related systems, such as *Distrinet* [8], use a distributed approach across multiple computers to improve scalability, but the drawbacks remain similar to those of *Mininet*.

One system that stands out in particular and uses a similar approach as *Distrinet* is the testbed from Windisch *et al.* [13]. This testbed system also uses container virtualization distributed across multiple computers, but it allows real hardware and, therefore, possibly also emulators to be integrated. Unlike in *Mininet*, functions are also available that can automatically conduct experiments in the testbed. In addition to the fact that an isolated kernel is not available to every host in this testbed system, the focus on SDNs in particular provides functions that unnecessarily complicate the use for evaluating end-to-end security protocols.

C. Infrastructure and Topology Managers

OpenBACH [9] is a tool that orchestrates the configuration and experiments on various hosts. These hosts can be physical or virtual but are, in any case, completely isolated from each other, which allows the evaluation of security protocols that require their own kernel modules. However, all hosts must be preconfigured, and the topology between the hosts cannot be configured dynamically. Emulation tools can still be integrated, as the *OpenBACH* developers demonstrate using the example of the Geostationary Orbit (GEO) satellite simulator *OpenSAND* [14]. *OpenBACH* also offers a wide range of functions to automatically perform a variety of network experiments, e.g. using real web-server implementations, and to automatically collect the result data. However, *OpenBACH* is not a viable option for our testbed, as the hosts cannot be easily reset after experiments, which could affect the reproducibility of experiments. There has also been no active development for some time.

The *VITO* testbed system, which uses VMs with virtual network topologies and focuses in particular on link-emulation [10], comes very close to our defined requirements. Unfortunately, as in many cases, the source code is not publicly available.

IV. Testbed Design & Implementation

As we could not find a testbed system that completely fulfilled our requirements, we started to developed our own solution with *Proto²Testbed*. It aims to provide a system that makes it easy to evaluate security *proto*cols and support developers during *proto*typing of new implementations. As *OpenBACH* and *VITO* fulfill many of our requirements, we have adopted some concepts in the design of *Proto²Testbed*.

The proposed testbed system uses the virtualization framework *QEMU* [15] to create isolated VMs using hardware-virtualization with the Kernel-based Virtual Machine (KVM), called `Instances`. The *Proto²Testbed* software is installed on a Linux workstation or server and, after minimal manual preparation, independently takes care of setting up the testbed's `Instances` and network topology, carrying out experiments, and shuts down the testbed at the end.

A. Structure and Components

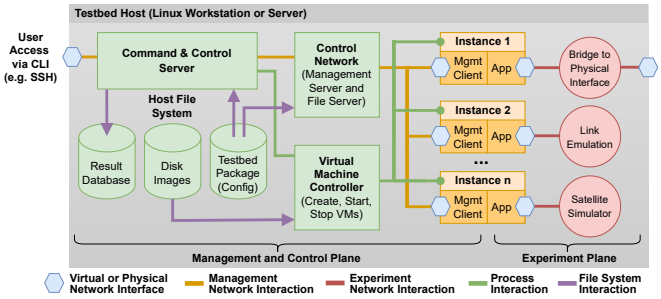


Fig. 2: Overview of the components of *Proto²Testbed*.

An overview of *Proto²Testbed* is provided in Figure 2. A central component of the testbed system is the Command and Control Server, a Command Line Interface (CLI) tool written in *Python 3*. The computer, which this tool and its dependencies are installed on, is called the `Testbed Host`. The user invokes the testbed system with a path to the `Testbed Package`, which contains the configuration for the testbed execution. The execution of the testbed is fully automated, but the user can pause the execution at various points to manually access the `Instances`. The Command and Control Server starts the `Instances` using a prepared VM base disk image, and the initial config of the `Instances`, e.g. setting up basic networking and a hostname, is handled using *cloud-init*. The disk images are used in a copy-on-write mode, so any changes made to the file system of an `Instance` are only temporary and do not affect the underlying disk image. The preparation of such a base disk image is described in Section IV-B.

1) Testbed Package: Except for the disk images, the `Testbed Package` contains all files that are required for a testbed. The structure of a testbed, e.g. `Instances`, network topology, and experiments, is defined in a testbed configuration file, see Listing 1 for an example. At least this JavaScript Object Notation (JSON) configuration must be included in a `Testbed package`.

Alongside the configuration, additional scripts and dependencies are contained in the Testbed Package. These include setup scripts, which are downloaded to the Instance and executed upon the start, and additional programs and dependencies for the Instances that are installed by the setup script. The contents of the Testbed Package are made available to all Instances with a file server that the Command and Control Server manages. In the best case, all files of a Testbed Package are kept in a version control system, allowing multiple users to work together on testbed setups.

```
{
  "settings": {...},
  "networks": [{
      "name": "exp0",
      "host_ports": []
    }],
  "integrations": [...],
  "instances": [{
      "name": "client",
      "diskimage": "base_image.qcow2",
      "setup_script": "client/setup.sh",
      "environment": {"SERVER": "10.10.0.1"},
      "networks": ["exp0"],
      "applications": [...]
    },{
      "name": "server",
      "diskimage": "base_image.qcow2",
      "setup_script": "server/setup.sh",
      "environment": {"CLIENT": "10.10.0.2"},
      "networks": ["exp0"],
      "applications": [...]
    }]
}
```

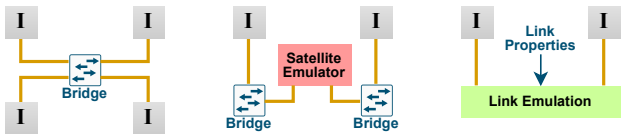
Listing 1: Minimal example of a testbed configuration with two Instances.

2) Instances: On all Instances, a Management Client needs to be installed, which is responsible for communication with the Management Server, which is in turn provided by the Command and Control Server on the Testbed Host. It receives commands, like setup instructions or signals to run experiments from the Management Server, and reports the current status of the Instance back. Apart from this communication, the Instance can download files from the file server on the Testbed Host.

3) Network Topology: The aforementioned communication is handled in a separate management network, which is created upon testbed start using Linux bridges and connects the Testbed Host and all Instances. The Instances can access the Internet (e.g. for downloading additional dependencies) via this network.

The user defines at least one experiment network in the Testbed Package for the actual experiments. Each Instance will be connected to one or more of these experiment networks. For that, it gets a *virtio* para-virtualized network interface[1] for each connected network. On the Testbed Hosts, these connections are realized with Linux bridges [16], that ultimately act like a layer two network switch.

[1]The *virtio* interface allows up to 10 Gbit/s connection speeds. We assume that this is plenty for the test of single end-to-end connections via satellite networks.



(a) Connection via Bridge ("Switch") (b) Connection via Emulation Tool (c) Connection via Link Emulation

Fig. 3: Three possible concepts for network topologies between Instances (I = Instance).

It is possible to add physical network ports of the Testbed Host to an experiment network. Some network simulators can interact with the real world using TAP-devices, which can also be integrated into an experiment network via the bridges. In Section VI we further describe possible options for integrating simulators for satellite mega-constellations. Figure 3 shows examples of network topologies that can be created with *Proto²Testbed*. In addition to the topologies shown in the Figure, real network paths can be integrated by connecting physical network ports of the Testbed Host to a bridge.

4) Applications and Experiments: Experiments are conducted by running special programs on the Instances, called Applications. These Applications are deployed together with the Management Client and usually collect data during the experiments; their results are exported in real-time to an *InfluxDB* time series database via the management network. The testbed configuration defines, which Application runs on each Instance. Individual settings can be defined in the configuration for each Application, whereby any number of Applications can run in parallel on an Instance. An example for an Application is a wrapper for the well-known speed test tool *iPerf3* [17]. In addition to Applications, it is also possible to run manual experiments by executing scripts on the Instances, that are, for example, downloaded from the Testbed Package or provided in other ways. The Management Client of *Proto²Testbed* provides an interface that allows users to develop their own Applications.

5) Integrations: Whenever a testbed requires an automated action on the Testbed Host itself, such as the start of an emulation tool or configuration changes to a network interface, this is carried out by Integrations. An example how such an Integration can be used is shown in Section V-B. *Proto²Testbed* provides multiple types of Integrations; one example is a program that just invokes a script from within the Testbed Package. Several Integrations can be executed in parallel; the settings and also the phase of the testbed execution, at which the Integration is to be invoked, are defined in the testbed configuration. An interface allows users to develop their own Integrations.

B. Testbed Workflow

A typical *Proto²Testbed* workflow is shown in Figure 4; the following steps are necessary, whereby only a few have to be conducted manually:

1) The OS, e.g. Debian Linux, is installed to a new disk image. This step is required only once in most cases.
2) The Management Client (with all available `Applications` and required dependencies) is installed to the image. In this step, dependencies that are required for the end-applications or protocols that will be evaluated but do not change during development (e.g. shared libraries) can also be installed, saving time during later testbed startups. For semi-automatic handling of this step, additional tools are provided by *Proto²Testbed*. The base disk image that was created after finishing this step is used only in a read-only way by the testbed system – it is not changed and can therefore be used across multiple testbed executions and shared by concurrent `Instances`.
3) During startup, the testbed system creates `Instances` based on these base disk images. The initial configuration of the `Instances` is done via *cloud-init*. When this step is completed, the `Instance` is up and running, and the Management Client is connected to the Management Server.
4) The Management Server sends experiment-specific setup instructions to all `Instances`. They can download setup scripts that are contained in the `Testbed Package`. The setup script can, for example, change settings for the experiment network interfaces or start a Virtual Private Network (VPN) tunnel software to other `Instances`. Also, additional assets, e.g. programs containing an implementation of a security protocol that will be evaluated in the testbed, can be downloaded by the setup script from the `Testbed Package` to the `Instance`. After this step is completed, all `Instances` of a testbed should be ready to start the actual experiments.
5) After the experiments are completed, all results are exported to the `Testbed Host`, where they are stored in a persistent way. The testbed is then dismantled. The Command and Controller Server ensures that the state of the `Testbed Host` is fully reset after a testbed execution has been completed. After that, a new testbed execution can follow, e.g. based on the same base disk images but with a changed configuration.

C. Result Extraction

Since all `Instances` are destroyed after a testbed execution is finished, the results of the experiments need to be stored elsewhere. For results exported by `Applications`, data is instantly written from the `Instances` to an *InfluxDB* database running on the `Testbed Host`. Results can be visualized in real-time using *Grafana*, which is particularly helpful during debugging. *Proto²Testbed* also provides additional scripts for exporting the results of a testbed execution from the database to images or CSV files so that users can use their own workflow for evaluation.

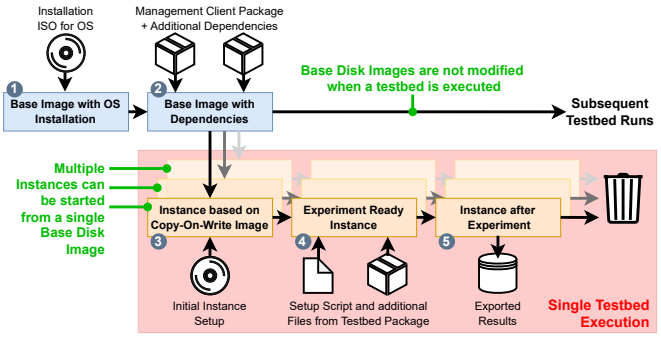


Fig. 4: Overview of the required preparation work and dependencies for a testbed and the lifecycle of a single testbed execution.

When specified, files are downloaded from the `Instances` after the testbed execution. This could be the case, for example, after the manual execution of scripts without specially developed `Applications` inserting result data to the InfluxDB, or when packet-traces are recorded during experiments. In that case, these files are copied to the persistent file system of the `Testbed Host`. These approaches offer a high degree of flexibility to use the testbed system for various applications and allow users to integrate it into existing tools or workflows for analysis.

D. Integration into CI/CD

Proto²Testbed can be integrated with little effort into existing Continuous Delivery/Integration (CI/CD) tools, such as these provided by *GitLab*, to facilitate automatic testing of end-to-end protocols. With placeholders, which can be used anywhere in the testbed config, the testbed can be dynamically adjusted with environment variables so that, for example, different network conditions or protocol settings can be tested quickly without having to write separate testbed configurations for each test case.

V. Usage Examples

We have already developed a fully functional prototype of *Proto²Testbed*, allowing us to present initial experiments. This section demonstrates two example testbed setups and outlines the performance of the testbed system. We publicly provide the source code of *Proto²Testbed*, documentation, and testbed configs for the examples under the GPLv3 license so that results can be easily reproduced.[2] These examples are also a great way to get started with the usage of the testbed system.

A. Simple Topology

This simple testbed, which is shown in Figure 5, consists of three `Instances` and two experimental networks. The first `Instance` acts as a client and hosts two `Applications`: an *iPerf3* client and a ping `Application`. The third

[2] https://github.com/martin-ottens/proto2testbed

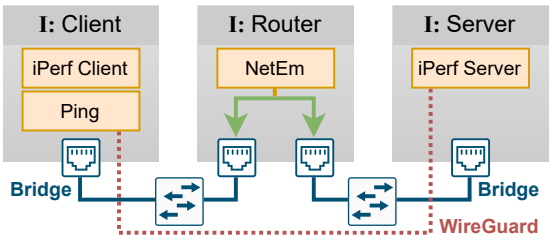


Fig. 5: Example testbed with a simple topology. Orange boxes are `Applications`, that are fully managed by the testbed system.

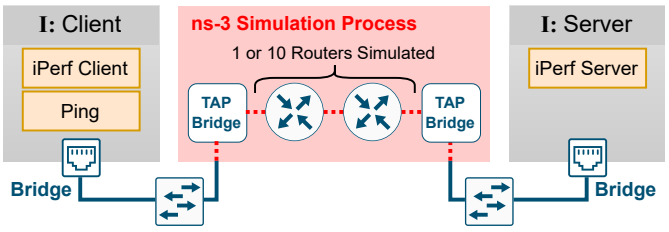


Fig. 7: Integration of a simulator into a testbed to provide an emulation environment. The simulator provides a virtual network topology and is fully managed by the testbed system.

`Instance` hosts the corresponding *iPerf3* server. Client and server communicate via the second `Instance`, which acts as a router between the two experiment networks.

During the experiment, an `Application` is executed on the router `Instance` that increases the Round Trip Time (RTT) from client to server to around 100 ms. Optionally, a *WireGuard* VPN tunnel can be created between the client and server `Instance`, providing two testbed setup variations that can be compared: Unencrypted communication versus communication via a security protocol.

Once the `Testbed Package` is created, the whole testbed execution is automated. The automated execution includes managing of all involved `Applications`, the data collection, and the rendering of the result plots.

Both testbed variations were run consecutively; the combined results are shown in Figure 6. A maximum of 10 Gbit/s can be achieved, which is limited by the network interface of the `Instances`. The goodput decreases when the VPN tunnel is used. As expected, the increasing latency at time $t = 30s$ has a negative effect on the throughput in both variations.

Proto²Testbed not only offers the ability to perform such tests automatically, but it is also effortless to test and compare different parameters, such as different delays in case of this example. With variables in the testbed config, different testbed variations can be tested without the need to change the configuration.

This example demonstrates, that *Proto²Testbed* can be used to evaluate the performance of end-to-end security protocols. The implementation of further `Applications` that can collect additional metrics, i.e. for evaluating the Quality of Experience (QoE), is subject to future work.

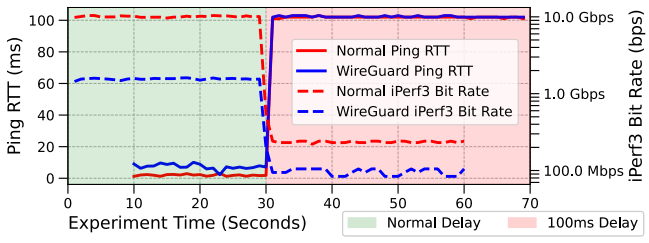


Fig. 6: Bandwidth and ping measurement results from both testbed variations of the first example. At $t = 30s$, the router introduced an artificial delay.

B. Emulation Environment

The topology of the second example is shown in Figure 7 and integrates a simulator into the testbed. As in the first example, two `Instances` are representing a server and client and hosting the same `Applications`. The router is replaced with an *ns-3* simulation process that runs in real-time mode and uses the *tap-bridge* function of *ns-3* [2] to generate two TAP-interfaces on the `Testbed Host` that are connected to the experimental networks.

In the *ns-3* process, simple Local Area Networks (LANs) are simulated that contains the client and server TAP bridge nodes. The TAP-interfaces on the `Testbed Host` are connected to these nodes and a configurable number of routers connects the nodes, providing a virtual network topology. This example also has two scenarios: The testbed is executed once with one router and once with ten routers between the two simulation nodes. The *ns-3* process is fully managed by an `Integration`.

In this example, *Proto²Testbed* provides a fully integrated emulation environment. The results of the *iPerf3* bandwidth measurements and ping tests conducted in both scenarios are visualized in Figure 8. It becomes clear how much the simulator becomes a bottleneck in such a setup. Even with one router, the throughput drops massively compared to the throughput of the first example. The poor performance is mainly observable due to the delay introduced by the simulator. In the scenario with ten routers, simulation events are already significantly

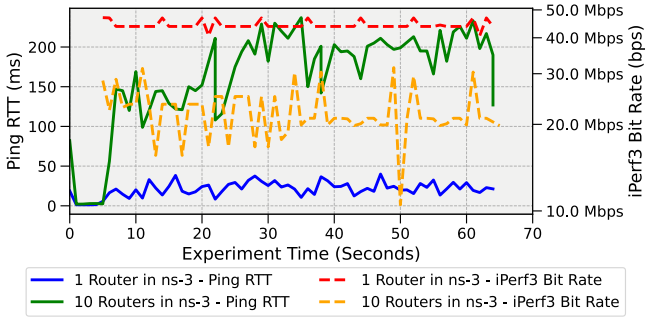


Fig. 8: Bandwidth and ping measurements during communication via a simulated topology with one or ten routers.

delayed, which causes the large fluctuations and indicates that the simulator is overloaded.[3] Analyzing how realistic these results are and if such an emulation environment can be scaled for large satellite constellations is subject to further work. Section VI-A2 already describes some approaches that we are actively investigating in order to be able to provide a scalable emulation environment in the future.

In both examples, the whole testbed execution took around 110 seconds on a modern workstation, of which around 40 seconds were used to start, prepare and stop the testbed. Another example that integrates real hardware into a testbed can be found in the provided example source code.[2] This shows, that *Proto²Testbed* could also be used to conduct tests via real-world satellite networks.

VI. Discussion & Further Work

Even though many functions have already been implemented in *Proto²Testbed*, there are possible improvements to allow the testbed system to be used comprehensively for testing end-to-end security protocols via satellite constellations.

A. Emulation of Satellite Constellations

The main focus is on developing a suitable emulation that makes it appear to the `Instances` as if they are communicating via satellite paths of a mega-constellation. We will evaluate two different approaches for this in the future.

1) Integration of a Satellite Simulator: We intend to modify a simulation framework like *Hypatia* [11] so that the traffic of the `Instances` is routed through the simulator using virtual network interfaces. However, as described in Section V-B, it is not evident whether the simulator can compute constellations with hundreds or even thousands of satellites and complex routing algorithms in the required real-time mode and achieve sufficient traffic throughput at the same time. Another challenge with this approach is the generation of background traffic, which can be interesting for the behavior of the end-application under test but would significantly increase the overall traffic volume that has to be handled by the simulator. We assume this approach is only feasible for small constellations and end-applications that generate limited amounts of traffic.

2) Link Emulation: The second approach is more complex to implement but will most likely allow a considerably better scalability. A simulation that does not run in real-time generates traces for pre-selected link pairs, e.g. a ground station and a user terminal. These traces include various properties of the links, such as delay, jitter, bandwidth, and packet loss. Recorded traces will have a specific update resolution of e.g. $100\,\mathrm{ms}$. The simulation considers all factors of the constellation, such as satellite positions, routing, background traffic, and mobility – these factors are ultimately reflected in the traces. Different traces are generated for different scenarios in the simulation.

These traces are pre-generated and ”replayed” by a link-emulation tool during the testbed execution between two `Instances`. For the end-applications and protocols installed on these `Instances`, it then should appear as if they were communicating via the previously calculated satellite link. This split approach not only moves the compute-intensive simulation away from the testbed execution but also offers some additional advantages, such as the fact that the same trace can be used to compare different protocols for good comparability. Also, these traces can be created using different simulation tools from various research areas, given that the emulation tool will only require a specific format for the trace files.

There are already approaches to the required link-emulation tool, as the work of Stockmayer *et al.* [10] and Tian *et al.* [18] shows. We will initially pursue an approach with *NetEm*. Alternatively, such a link emulation can be implemented with a simulation tool such as *ns-3*, which plays back the traces in a real-time mode as part of a channel emulation. Nevertheless, additional verification of this approach is required, for which existing simulation tools can provide assistance.

The fact that the testbed system initially considers the network abstractly means that various emulation tools, as well as real hardware, can be flexibly integrated.

B. Further Improvements to the Testbed System

Other improvement options concern the handling of the testbed system itself. Many of these improvement options have resulted from tests of the testbed system that have been conducted by users, whereby different users may use the testbed system for different tasks.

This includes the following features:

- Currently, all `Applications` and their dependencies are bundled together with the Management Client and must be installed together to a base disk image. It is impossible to dynamically add new `Applications` via the Management Client at runtime. We envisage an option to deploy new `Applications` to the `Instances` during testbed executions, e.g. during the `Instance` setup.
- It is required to log in to an `Instance` using Secure Shell (SSH) via the Management Network whenever a user wants to interact with the `Instance`. Other, more user-friendly options can be added for that access, e.g. by adding an interactive CLI frontend to *Proto²Testbed*, which users can use to connect to the `Instances` via a serial console, for example. This also allows `Instances` to be debugged entirely without a network connection.
- Due to system restrictions, only one testbed can currently be executed on a `Testbed Host` at any given time. A modern multicore workstation can easily accommodate several concurrent testbeds without interference

[3] *ns-3* was executed in a *BestEffort* mode in this example. Alternatively, a *HardLimit* for event delays can be configured, which causes the simulator to be terminated in such overload cases. This prevents falsified testbed results being generated by the unwanted simulator delays up to a certain point.

due to resource constraints; it therefore is beneficial to adapt *Proto²Testbed* to allow parallel executions. Integrating additional features, such as CPU Pinning of the `Instances`, could also help minimize interference from concurrent testbeds.

We assume that there is even further potential for improvement with future use of the testbed system in research projects, particularly to cover various different workflows. Any updates will be provided to our repository.[2]

VII. Conclusion

There are numerous approaches to the realization of network testbeds. When testing real applications or protocols, many researchers and developers, especially in the field of satellite communication, nevertheless resort to project-specific testbeds that are not universally applicable or require a lot of maintenance and administration work. After defining our requirements for a testbed system that can be used to test the behavior of real application and protocol implementations via satellite mega-constellation networks, we reviewed several existing testbed systems. Since no system fully fulfilled our requirements, we decided to develop our own testbed system.

With *Proto²Testbed*, we laid the foundation for an automated and integrated testbed system. Users have to make minimal manual preparations and define the structure of their testbed, the configuration of the hosts, and the experiments to be executed in the form of a `Testbed Package`. The testbed system automatically takes care of providing VMs that are hosting the end-application and security protocols under test. The testbed system connects them via a virtual network topology, orchestrates pre-defined or manual experiments, and finally collects the results.

We have shown in two examples how *Proto²Testbed* can run experiments and what kind of performance a user can expect. In particular, a simulator can already be integrated into the topology to provide an emulation environment.

For the planned use in research projects, some further challenges still need to be solved, especially the development of a scalable solution for link-emulation. We have presented two approaches to these challenges that we want to implement and evaluate in future work. *Proto²Testbed* should become a valuable tool in testing and developing security protocols used in satellite networks – and potentially even in other areas.

Acknowledgments

This work was carried out under ESA Contract No AO/1-11869/23/NL/AF under the Element 1 of the ESA Programme Related to EU Secure Connectivity and funded by the European Space Agency. Through this ESA Programme, ESA contributes to the definition, development and validation of the EU Secure Connectivity governmental infrastructure under the Union Secure Connectivity Programme of the European Union (IRIS2). The view expressed herein can in no way be taken to reflect the official opinion of the European Space Agency.

References

[1] J. A. Donenfeld, "WireGuard: Next Generation Kernel Network Tunnel." in *NDSS*, 2017, pp. 1–12.

[2] nsnam, "ns-3 Wiki – HOWTO make ns-3 interact with the real world," 2017. [Online]. Available: https://www.nsnam.org/wiki/HOWTO_make_ns-3_interact_with_the_real_world

[3] J. Gomez, E. F. Kfoury, J. Crichigno, and G. Srivastava, "A survey on network simulators, emulators, and testbeds used for research and education," *Computer Networks*, vol. 237, p. 110054, Dec. 2023. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1389128623004991

[4] D. Camara, H. Tazaki, E. Mancini, T. Turletti, W. Dabbous, and M. Lacage, "DCE: Test the real code of your protocols and applications over simulated networks," *IEEE Communications Magazine*, vol. 52, no. 3, pp. 104–110, 2014.

[5] T. Werthmann, M. Kaschub, M. Kühlewind, S. Scholz, and D. Wagner, "VMSimInt: a network simulation tool supporting integration of arbitrary kernels and applications," in *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools '14. Brussels, BEL: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014, p. 56–65. [Online]. Available: https://doi.org/10.4108/icst.simutools.2014.254623

[6] D. Anderson, M. Hibler, L. Stoller, T. Stack, and J. Lepreau, "Automatic Online Validation of Network Configuration in the Emulab Network Testbed," in *2006 IEEE International Conference on Autonomic Computing*. Dublin, Ireland: IEEE, 2006, pp. 134–142. [Online]. Available: http://ieeexplore.ieee.org/document/1662391/

[7] M. P. Contributors, "Mininet – An Instant Virtual Network on your Laptop (or other PC)," 2022. [Online]. Available: https://mininet.org/

[8] G. Di Lena, A. Tomassilli, D. Saucez, F. Giroire, T. Turletti, and C. Lac, "Distrinet: a mininet implementation for the cloud," *SIGCOMM Comput. Commun. Rev.*, vol. 51, no. 1, p. 2–9, Mar. 2021. [Online]. Available: https://doi.org/10.1145/3457175.3457177

[9] Viveris Technologies and CNES, "OpenBACH – Overview," 2023. [Online]. Available: https://www.openbach.org/overview.html

[10] A. Stockmayer, C. Kindermann, and M. Menth, "VITO: VIrtual Testbed Orchestration for Automation of Networking Experiments," in *Proceedings of the 11th EAI International Conference on Performance Evaluation Methodologies and Tools*, 2017, pp. 113–118.

[11] S. Kassing, D. Bhattacherjee, A. B. Águas, J. E. Saethre, and A. Singla, "Exploring the "Internet from space" with Hypatia," in *Proceedings of the ACM Internet Measurement Conference*, ser. IMC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 214–229. [Online]. Available: https://doi.org/10.1145/3419394.3423635

[12] T. Schubert, L. Wolf, and U. Kulau, "ns-3-leo: Evaluation Tool for Satellite Swarm Communication Protocols," *IEEE Access*, vol. 10, pp. 11 527–11 537, 2022.

[13] F. Windisch, K. Abedi, T. Doan, T. Strufe, and G. T. Nguyen, "Hybrid Testbed for Security Research in Software-Defined Networks," in *2023 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, Nov. 2023, pp. 147–152. [Online]. Available: https://ieeexplore.ieee.org/document/10329614/

[14] E. Dubois, N. Kuhn, J.-B. Dupé, P. Gélard, F. Arnal, C. Baudoin, A. Delrieu, and D. Pradas, "OpenSAND, an open source satcom Emulator," *Proc. Kaconf*, pp. 1–4, 2017.

[15] F. Bellard, "Qemu, a fast and Portable Dynamic Translator," in *USENIX annual technical conference, FREENIX Track*, vol. 41, no. 46. California, USA, 2005, pp. 10–5555. [Online]. Available: https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf

[16] N. Varis, "Anatomy of a Linux bridge," in *Proceedings of Seminar on Network Protocols in Operating Systems*, vol. 58, 2012, pp. 58–63. [Online]. Available: https://core.ac.uk/download/pdf/301129313.pdf#page=62

[17] ESnet/Lawrence Berkeley National Laboratory., "iPerf3: A TCP, UDP, and SCTP network bandwidth measurement tool," https://github.com/esnet/iperf, 2014, available at https://iperf.fr/.

[18] W. Tian, Y. Li, J. Zhao, S. Wu, and J. Pan, "An eBPF-Based Trace-Driven Emulation Method for Satellite Networks," *IEEE Networking Letters*, pp. 1–1, 2024.

All links were last accessed on January 11, 2025.