coreemu-lab: An Automated Network Emulation and Evaluation Environment

Lars Baumgärtner*, Tobias Meuser†, Bastian Bloessl*

Technical University of Darmstadt, D-64289 Darmstadt, Germany
*FB 20, E-mail: {baumgaertner, bloessl}@cs.tu-darmstadt.de
†Multimedia Communications Lab, FB 18, E-mail: {tobias.meuser}@kom.tu-darmstadt.de

***Abstract*—Developing new solutions for challenging communication environments requires extensive testing in various different scenarios. While discrete-event network simulators scale well, their main use is in evaluating specific algorithms and not necessarily assessing the real-world performance of an actual application. In contrast, full network emulation is a costly but more realistic way to perform an evaluation. These types of evaluation provide more valuable insights for both developers and users. Nevertheless, oftentimes the wheel is reinvented to provide a newly written simulation environment for a specific networking software or handwritten evaluation and reporting scripts for common metrics. Unfortunately, the downside of customized solutions is potential oversights regarding important metrics, and later finding out various problems when deploying the software in the field. In this paper, we present *coreemu-lab*, a novel framework that automates the process of orchestrating different monitoring services in an emulated, lightweight networking environment. This includes simulated mobility, automated data collection, and analysis of common metrics. It is specifically designed for the evaluation of resilient, decentralized communication software in challenging, mobile scenarios. Our evaluation shows the flexibility and ease of use of our proposed multi-platform, open-source solution.**

***Index Terms*—Network Simulation, Network Emulation, Automated Evaluation, Disruption-Tolerant Networking.**

I. Introduction

Complex communication scenarios involving mobile nodes at various speeds (e.g., pedestrians, UAVs, and cars) or failing communication infrastructure during natural disasters pose great challenges to networking software (Fig. 1). Thus, thorough testing of such applications in various scenarios is necessary. This is especially important for software that should be used in emergency scenarios and networks such as mesh and disruption-tolerant networking (DTN). There are many different solutions to simulate mobility and evaluate networking protocols. They are, however, often complex to use, limited in scope, and tailored to the specific needs of research communities. Furthermore, they often lack general reporting functionality, only perform algorithmic simulations, or require custom simulation models, specific for the simulator. Ideally, one wants to run and evaluate the actual application without reimplementing the algorithm in Java or C++, as required for *The ONE* [1] or *OMNET++* [2], respectively. Prior research in the field of emergency communication and DTNs [3], [4], [5], [6] often contains similar evaluations of different software. Yet, they do not share a common platform

Fig. 1: Example scenario involving various mobile nodes with different movement speeds and communication ranges.

for result collection and evaluation, requiring the researcher to set up the process from scratch for every project. This might even include standard metrics such as resource utilization (CPU, memory, bandwidth, etc.) when evaluating a DTN stack for a disaster messaging service. This leads to the fact that some side effects of a concrete implementation are not directly visible and, thus, may be overseen in the analysis. By automating this process and making it easy to extend and use, the quality of network evaluations for challenging network conditions in general can be improved. Also, besides supporting researchers in their evaluation, this can also be used by developers in Continuous Integration (CI) services to ensure correctness and performance in a predefined set of scenarios. Additionally, this provides non-domain experts the opportunity to evaluate and compare different software in a standardized way and compare performance metrics. This also leads to the opportunity to share a standardized set of tests and evaluations for emergency communication scenarios.

In this paper, we present *coreemu-lab*[1], a novel open-source

[1]https://github.com/gh0st42/coreemu-lab

framework that streamlines the process of setting up, running, and evaluating of complex network simulation scenarios, combining network simulation, system monitoring and analysis tools. To this end, we created a self-contained and portable system, wrapped into a docker container. Thus, except for docker, no further dependencies are needed on the host. Our solution is built upon the *CORE network emulator (coreemu)* [7] and can be run interactively with a GUI or headless for automated experiments. Through *coreemu*, a network topology can easily be created interactively. Furthermore, movement patterns can be added to simulate node mobility. We provide a wide range of monitoring services that work independent of the simulated software and gather metrics such as network utilization, CPU and memory usage as well as wireless contact times throughout the simulation. The emulated nodes are all isolated in Linux namespaces and thus can run any Linux software, e.g., IBR-DTN [8] or dtn7 [9]. Finally, we collect all relevant data from the emulated nodes and provide configurable reports and figures from these metrics. The whole system is designed to be customized for third-party sensors and analyzers.

In this paper, we present the following contributions:

- A new integrated, multi-platform, and batteries-included environment for interactive emulated network services
- A novel framework for fully automated network simulations, including reporting
- A set of command-line tools to orchestrate and automate *coreemu*
- A novel tool for interactive movement pattern creation
- A novel rust crate to interact with *coreemu* for further automation and utility development

II. Related Work

There are many different environments for network simulations [1], [2], [7], [10], [11], [12] of which some also include basic reporting functionality.

Commonly used tools such as *The ONE* [1], *OMNET++* [2], and *ns3* [10] are all discrete-event network simulators. Therefore, they scale really well for large-scale simulation but usually only implement specific protocols within their environment. Thus, one has to write code specifically for the simulator in a predetermined language, e.g., Java or C++. These systems are not meant to run off-the-shelf networking code as they would be deployed on an actual system.

Some of these tools are highly domain-specific, such as *The ONE* [1], which is solely designed for DTN simulations and the evaluation of different DTN routing algorithms. Regular communication or mesh routing is not possible within this environment. But for its intended purpose, it already comes with many movement models and a toolkit for generating reports and plots. As a result, it is very easy to work on new DTN routing algorithms and get an instant evaluation and comparison with the standard algorithms shipped. Therefore, it is very popular within the research community providing add-ons such as natural disaster mobility models [13] or using them for information-centric networking (ICN) in disaster situations [14].

For the most realistic network simulation, setups like *MiniWorld* [15], that use full system emulation with QEMU instances for the nodes, can be used. While this has the benefit of being able to simulate different CPU architectures running completely different operating systems, starting such a simulation takes a long time and is also very heavy on the host system resources, even when distributed amongst different machines. Thus, quickly simulating a larger number of nodes is infeasible.

Other approaches such as QOMB [16] which were used in DTN evaluations [17] try to achieve this scalability by using massive amounts of physical computers, which is not practical for most use-cases.

As a more resource preserving alternative to full system emulation, a few solutions such as *Mininet-WiFi* [11], *coreemu* [7] and *meshnet-lab*[2] make use of Linux network namespaces to separate userland programs. Therefore, the simulation is as lightweight as starting the actual programs of each node plus some management overhead for mobility and link calculation. While the first one mainly focuses on Software-Defined Wireless Networks (SDWN), the last one is very young and only provides basic features. As *coreemu* has been around for quite some time, is actively maintained, and also has integration for advanced radio-link emulation using *EMANE* [18], it was chosen as the basis for *coreemu-lab*. Apart from more realistic propagation models, *EMANE* also allows integration of Software-Defined Radios for hardware-in-the-loop experiments with custom physical layers. Its main drawbacks are that it is not very portable, requires quite a few dependencies, and lacks good built-in report generation.

Another related field is automated network experiments. Here, Frömmgen et al. [19] presented *MACI*, a very powerful platform for automatically performing large numbers of experiments with integrated results analysis. It consists of multiple docker images and a web frontend. *MACI* can work with different simulation backends, but it is not intended to be run interactively to watch or debug a single experiment. Also, users must provide their own post-simulation analysis scripts for generating the reports. While it definitely has its benefits when running large sets of experiments in more or less easy to parameterized configurations, e.g., 5 topologies x 5 random seeds x 5 different algorithms resulting in 125 individual runs, it also brings much complexity with it.

So the most commonly used software in research and industry often focuses on a single aspect such as simulating a network environment or they ship a rich environment, including plotting toolkits, but are very complex and even then do not work out of the box for the most common scenarios. Thus, we see a need for a portable, preconfigured and accessible solution for the evaluation of networking software that can easily be extended to ones needs.

III. Design

With *coreemu-lab*, we want to achieve two major goals: 1) an interactive tool for exploring and playing with network

[2]https://github.com/mwarning/meshnet-lab

applications and 2) an automated and scalable automation framework that works without human interaction. A few principles lead many of our design decision:

a) Convention over Configuration: We want the system to be easily accessible and of practical use. Thus, we choose sane and common defaults, which can be overridden but should work in most cases out-of-the-box. This includes the selection of gathered metrics but also the following report generation.

b) DRY: Don't Repeat Yourself: Testing and evaluating networking software thoroughly can be a very repetitive task. This includes selecting and starting monitoring sensors for various metrics but also includes the creation and selection of mobility models. Furthermore, writing scripts for plotting graphs and generating reports is a reoccurring and often very similar task, even for slightly different sensor metrics.

c) Batteries Included: Whether one is a researcher interested in improving specific aspects of, e.g., a DTN routing protocol, or a user trying to decide which application has the desired properties in a specific scenario, learning and adapting to new and complex tools is often very time-consuming. Therefore, common tools and frameworks should already be present and usable, so one can focus on the interesting aspects of such an evaluation.

d) KISS - Keep It Simple, Stupid: We think it is more beneficial in the long run if people do not have to learn too many new technologies for repetitive tasks such as network evaluations. Thus, we try to reuse tools and environments many people are already familiar with such as regular shell scripts for configuration and automation as well as python. Besides *gnuplot*, the latter is especially common for data science, analyzing evaluation results and generating meaningful figures. This should ensure that a broad number of users can use and easily extend *coreemu-lab* to fit their needs. Another aspect is that we want users to keep the freedom to implement their network services in whatever language/runtime they want. Our simulation environment can run anything that runs on stock Linux without modification, e.g., native binaries produced by Golang, rust, C/C++, JVM code or interpreted Python/JavaScript are all fine. There is no need to link to a specific simulation library or structure code differently.

e) Portability: We wanted an environment for our experiments that can easily run on different machines without the hassle of installing a million of dependencies that might not work on different Linux distributions or other operating systems. Everything should be as self-contained and portable as possible. Thus, one can easily move from a laptop for designing an experiment to big backend servers that are used for CI or large-scale simulation runs in parallel.

Figure 2 shows the main steps needed to perform a typical evaluation run. The remainder of this section describes the main components of this process in more detail.

A. Network Emulation and Mobility

As a foundation for the simulation of the network and the emulation of the different nodes, we selected *coreemu* [7]. This system is fairly lightweight, as each node is just a Linux namespace, separating it from all other nodes. Thus, the cost of running a node is only the processes contained in the actual namespace, the kernel and anything else needed is shared with the other systems. A full system emulation, e.g., using *QEMU* would provide even more flexibility and realism but does not scale as good and is very demanding on the hardware as shown by Schmidt et al. [15]. Additionally, support for different networking technologies such as wired links and various radio links is needed. While basic WiFi simulation is already possible in *coreemu*, more realistic models can be provided by *EMANE* [18], which also integrates nicely with the network emulator. Regarding the position of nodes, it is possible to define static network topologies or import ns2 movements that get replayed or looped for the simulation. The ns2 movement format is very common, used in various different simulation environments, and can also be converted to from bonnmotion [20]. Additionally, individual nodes can be assigned new positions during runtime through the GUI or the gRPC interface provided by *coreemu*. The actual wireless links are calculated and realized through local ethernet bridges and ebtables rules.

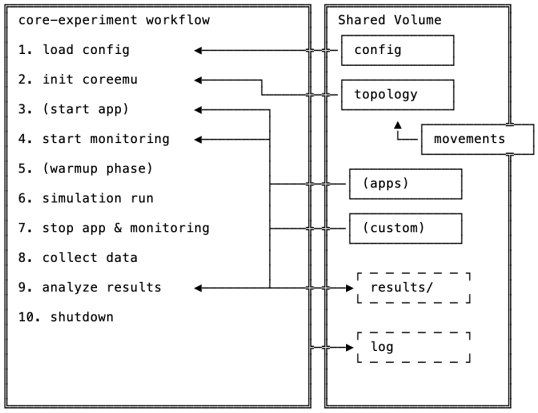


Fig. 2: Steps performed when running an experiment in *coreemu-lab*.

B. Monitoring Services

While some experiments require special sensors to evaluate certain aspects of the simulation, some standard properties are always of interest. The usage of local system resources such as CPU, RAM or I/O are always interesting, as they are a good indicator on the system requirements when deploying the application under evaluation. Furthermore, these can be used to draw conclusions on the energy consumption of the system. Additionally, network resources also have to be considered. Thus, logging data such as the used bandwidth on the different links at any given time can be very helpful. Beyond the costs and limitations of the network connection itself (e.g., expensive or volume-limited satellite or LoRa uplink), bandwidth can cost battery life. Another interesting property of the simulated system, especially for DTN emergency scenarios, is the number of contacts between the nodes and their duration.

All of these metrics can be gathered independent of a specific application running in the simulation.

Of course, custom logging for protocol specific monitoring services can easily be added without making changes to *coreemu-lab*. While extending our open source framework is always possible, it should not be necessary as third-party additions can be added through the experiment configuration.

C. Running the Experiment

In *coreemu-lab* there should be several modes of operation. Without any configuration the standard *coreemu* GUI is started, this way one can interactively design new network topologies and play with different node configurations. If a configuration is provided, one can choose between an interactive session with GUI or a headless version. Prior to starting an actual experiment, an optional warm up period should be configurable. Thus, nodes can already start moving and generate traffic or messages to have a realistic starting point. Without a configured runtime for the experiment, it will run until a user signal is received to properly end the experiment.

D. Data Collection

After the simulation has ended, the application(s) as well as the monitoring services are shut down. By default, all log files in any of the node subdirectories are collected and copied to a central results folder. To avoid any accidents or data loss when rerunning experiments, this folder should have a unique name. Thus, we construct it from the experiment name and a timestamp. Additionally, full copies of the node directories can be preserved in the results folder. As a lot of (binary) data can accumulate during a long-running simulation, this functionality is disabled by default.

E. Analyzing Results

Parsing different log files and drawing meaningful conclusions or plotting the results is often a tedious task. Even though the process is often very similar and the requirements are the same no matter what metric (CPU, RAM, etc.) is to be analyzed, there is no out-of-the-box program to do this. Often *gnuplot* or python with *pandas* and *matplotlib* are used for drawing graphs out of the raw numbers. While one might develop a good intuition by looking at the raw data pieces to spot different bottlenecks within an application, having automatically generate figures is often much easier to reason about. Therefore, we want standard reports that summarize an applications behavior over the simulated time. These analyses should give information on a per-node basis, with averages and standard deviation respectively the min/max values, but also give information about the overall simulation average of the specific metric. By also providing the standard data analysis frameworks within *coreemu-lab*, one can easily add custom reports using, e.g., Jupyter notebooks.

IV. Implementation

As some of the used tools have a lot of dependencies, which are not all available on all platforms or Linux distributions, we decided pack everything together in a docker container. Therefore, the whole setup is very portable, self-contained, and easy to deploy on new machines. An advantage of this setup is the increased reproducibility and that people can use our image as a basis for their own docker-based evaluation environments with other tools and features added. The whole system can be deployed by copying a single shell script that bootstraps the docker instance. We have successfully used this on various Linux distributions as well as macOS. All relevant setting can be provided through a shared directory, where also the results are logged and custom scripts are located. If our *clab* startup script is invoked without parameters and no *experiment.conf* is found in the default location, it just starts *coreemu* in an interactive session with its GUI. If an additional parameter after the shared volume is provided, an interactive shell is started. This can be used to replot some of the images or to test custom scripts without starting the *coreemu* GUI. A brief overview of the main components of the docker container can be seen in Figure 3. In the following, we give some more implementation details about the key components.

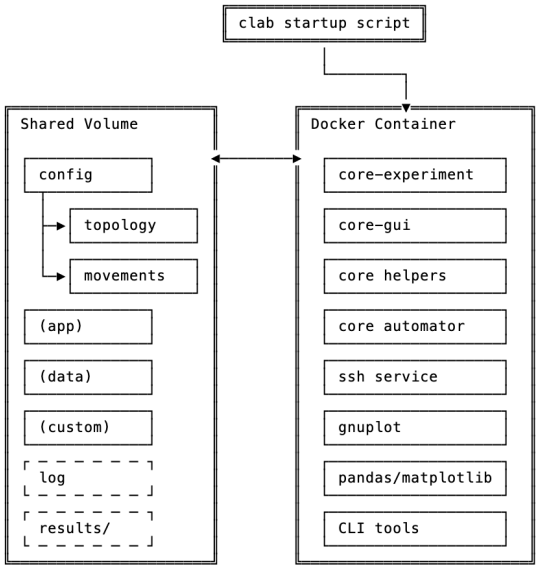


Fig. 3: General overview of *coreemu-lab*.

A. Experiment Configuration

The configuration file, *experiment.conf*, is actually a unix shell script that is loaded by the experiment runner (*core-experiment*). A full example of such a configuration can be seen in Listing 1. This approach provides a lot of flexibility, as one can easily add own variables or even have hooks with custom code that gets executed. By default, we support *pre*, *post* and *analyze* hooks that get automatically called (when present) at various stages of the experiment. These can be used to generate data on the nodes prior to the start of the simulation, perform custom processing after simulation has ended, or execute additional reporting scripts. The main downside of our approach is that shell scripts are rather strict

about their syntax when it comes to declaring variables and such. Thus, users need to be familiar with unix scripting.

```
NAME=example1
SCENARIO=twenty_nodes.xml

GUI=0

MONITOR_PIDSTAT=1
MONITOR_PIDSTAT_PARAMS="dtnd"

MONITOR_NET=1
MONITOR_NET_PARAMS="eth0"

MONITOR_CONTACTS=1
#MONITOR_CONTACTS_PARAMS="1"

MONITOR_XY=1
#MONITOR_XY_PARAMS="1"

#START_EXEC=("echo started > started.log"
# "echo second > second.log")

START_DAEMONIZED=('dtnd -n $(hostname)')

SHUTDOWN_PROCS="dtnd"

WARMUP=20
RUNTIME=300

#COLLECT_EVERYTHING=1

# Called prior to monitoring/app starting
pre() {
    echo "pre hook" > pre.txt
}

# Called prior to collecting logs
post() {
    echo "post hook - results in $1" > post.txt
}

# Called after logs have been collected
analyze() {
    echo "post hook - results in $1" > post.txt
}
```

Listing 1: *coreemu-lab* example configuration

B. Network Emulation and Mobility

As already mentioned, the main simulation is based on *coreemu* and, thus, requires Linux plus many dependencies to run. We put *coreemu* into a separate docker image[3]. By bundling it this way, it can be used on any platform that supports docker and X11 for displaying the GUI, e.g., macOS. We tried to keep this image as minimal as possible. Thus, the other *coreemu-lab* dependencies are added in another image[4] that is derived from the pure *coreemu* one.

To make working with *coreemu* from the command line and in scripts easier, we provide some helper scripts[5]. They contain functionality to quickly execute commands, in parallel or sequentially, on all nodes or daemonize a given application on the nodes. Additionally, they also contain functionality to generate random files and check for *FATAL* log entries in all running nodes, which usually indicate a crash.

Mobility can be achieved in several ways within the emulated network. With the GUI running, nodes can be interactively dragged around in real-time. For automated movement, one can also use the gRPC interface to directly update the positions of nodes in a running simulation. Through this interface, it is possible to also link external programs with live position data to the network within *coreemu-lab*. Finally, wireless nodes can be linked to *ns2* movement files for more complex mobility scenarios. This is especially useful as *bonnmotion* [20] can automatically generate various movement traces for this format and also convert to other formats.

To interactively change between different topologies and perform the transitions between them, we also provide a tool, the core-automator[6], to record and play back animations stop-motion-style. An example session with four wireless nodes can be seen in Figure 4. In the tool, you can specify a delay between each step, then drag the nodes around in the running simulation and record snapshots of these positions. Recording and playback can be initiated from the command line, but we also provide a graphical frontend for ease of use as seen in the figure. The file format of this recording is relative simple but also supports executing commands on nodes as displayed in Listing 2. This enables one to trigger events like message generations at specific points in time or at certain locations.

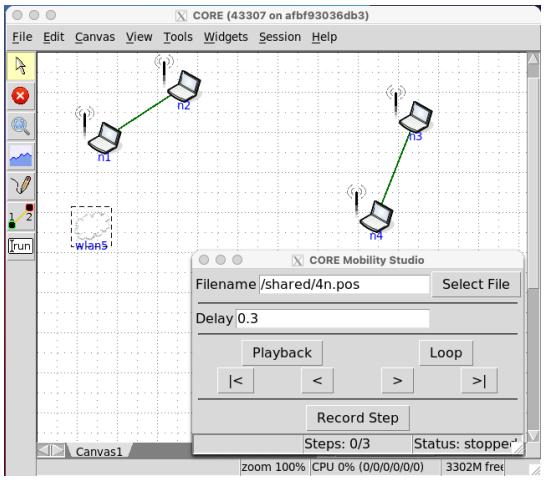


Fig. 4: *coreemu* with 4 nodes and our CORE mobility studio.

```
%delay 0.5

robot1 50 50 hostname > myhost
drone1 100 50
drone2 200 150
drone3 350 150
drone4 400 250
center1 763 276
```

[3]https://hub.docker.com/r/gh0st42/coreemu7
[4]https://hub.docker.com/r/gh0st42/coreemu-lab
[5]https://github.com/gh0st42/core-helpers
[6]https://github.com/gh0st42/core-automator

```
-- STEP
robot1 55 50
drone1 110 50
drone2 200 170
drone3 350 130
drone4 370 250

-- STEP
robot1 60 50
drone1 120 50
drone2 210 190
drone3 360 120
drone4 420 250
```

Listing 2: core-automator example scenario

While *coreemu* ships with a python API, which is not easily installable on non-Linux machines, we also developed a rust client library. This wrapper around the gRPC service makes it easy to get session and node information about a running simulation. Additionally, it can be used to perform updates to links and nodes from external tools. Having a statically compiled language, makes it easier to ship tools without requiring many dependencies and thus can be used on more platforms.

C. Monitoring Services

To monitor the running simulation, we rely mostly on small, resource efficient unix tools. Most information regarding a running process, such as CPU and memory (resident and virtual) as well as IO, can be easily collected through the *pidstat*[7] utility. By default, we monitor all processes on each node, but one can request that only a specific process, e.g., the main application, should be logged. As networking stats are vital for the evaluation of communication systems, we also collect network interface usage statistics through the use of *bwm-ng*. The emulated nodes can have more than one network interface. Thus, we log the traffic on all of them. If the amount of logging data is of concern, one can also restrict the monitored interfaces to only specific ones.

Especially for disaster communication where nodes are often very mobile and connections are disrupted frequently, it is important to know when and how often nodes had contact to others. Therefore, we wrote a monitoring service that directly gets radio-link and topology information from the running *coreemu* instance. These contact traces contain which node was in communication range of another at any given time. Furthermore, we also use this interface to record the exact position of a node at a specified interval. As movements can also be triggered through the external gRPC interface, having extra traces of the actual movement can be very helpful. This can also be used to correlate and interpret contacts and other network related events in the analyzes stage.

D. Running the Experiment

Prior to recording the actual experiment, an optional warm up period can be defined. This is commonly used to get applications in a realistic state where they already carry messages around and have gathered some routing information. After this step, the monitoring is started and the experiment will either run for a predetermined time or indefinitely, waiting for a manual shutdown. A manual end of the experiment can be triggered even in headless instances by creating a file *shutdown.txt* in the shared directory.

[7]https://github.com/sysstat/sysstat

TABLE I: Evaluation Scenario Settings

Scenario	Dimensions	1000 x 750 m^2
	Simulation time	240s
	Nodes	5
Mobility	Model	RandomWaypoint
	Speed	1.5 - 15 m/s
Traffic	Type	binary
	Interval	15 - 40 s
	Size	64 - 512 KB
App.	Name	goforban
	Discovery	5 s
	Routing	opportunistic/epidemic
Comm.	PHY rate	54 MBit/s (802.11g)
	Radio range	180m

E. Data Collection

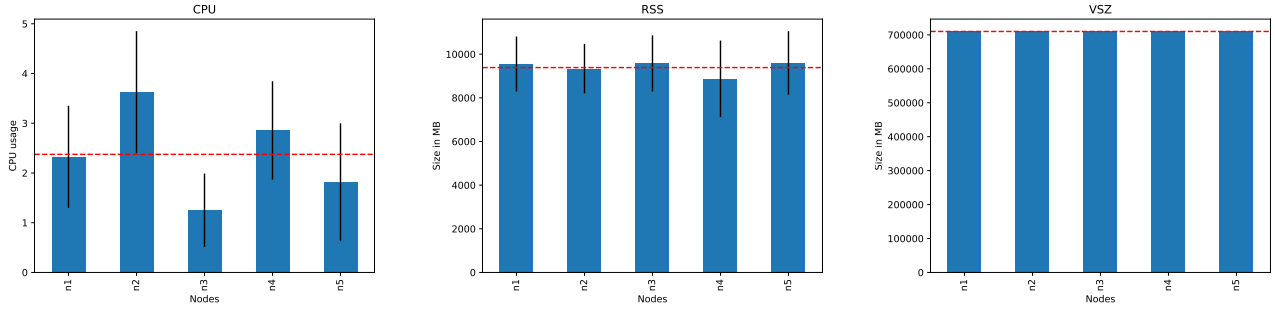
After the applications and monitoring services have been stopped that data from the emulated nodes is collected. By default, all log files are identified on the different nodes and copied to the results directory for further processing. If explicitly requested, the whole directory of each node can also be preserved containing everything stored during the simulation including large files and databases which are often not necessary for further analyzes.

F. Analyzing Results

In our standard setup, we generate plots of different metrics if monitoring was activated. We mainly use python with *pandas* and *matplotlib* for the visualization to output PDFs in the results folder. The pre-selected plots feature averages and standard deviation respectively minimum and maximum peaks for the nodes as well as the total mean across all nodes for a specific metric. Currently, plots are generated for various process related metrics, network usage, node movement and radio contacts. Additionally, textual statistics and reports about specific aspects of the simulation can be automatically generated, e.g., the contact times of wireless nodes or periodic message and file generation.

V. Experimental Evaluation

In the following, we present an experimental evaluation of some of the key components of our approach. All experiments were performed on a 4,2 GHz Quad-Core Intel Core i7 with 48 GB of RAM. The SSD drive should not play a big role as most parts of the simulation run on a *tmpfs* and thus reside in RAM. The system was running macOS 11.4 to show the portability and workflow of *coreemu-lab* on non-Linux platforms.



(a) CPU usage. (b) Residential memory usage. (c) Virtual memory allocated.

Fig. 5: Analysis of main process statistics gathered in the example simulation for each node as averages and standard deviation. The dashed red line represents the average across all nodes.

A. Scenario Setup

The details of our configured example scenario can be found in Table I. Our movement traces were generated through *bonnmotion* using a *RandomWaypoint* model and then converted to *ns2* for the inclusion in our topology file. Binary files are created in random intervals and shared amongst the nodes. The simulated adhoc WiFi emulates 802.11g representing older devices which usually have limited bandwidth. The network service we chose for our evaluation is an opportunistic gossip protocol called forban[8] that we implemented in Golang[9]. It epidemically replicates shared data when two nodes meet. A local neighborhood discovery is performed every 5 seconds.

B. Example Evaluation

In the following, we present various results as they are automatically generated by *coreemu-lab*. They are independent of any program specific parameters and should be applicable to any program under evaluation. These evaluations can be then combined with custom analyzers that, e.g., parse the log files generated by DTN daemons or mesh routing software.

To get a feeling for the performance of the analyzed app, we show a subset of the generated plots in Figure 5. Here we can see the average of any given metric over the experiment runtime for each node. Additionally, the standard deviation is plotted as error bars when applicable as well as the average across all nodes for the whole simulation is displayed as a dashed horizontal line in the plot. These types of plots can be used for several of the metrics.

Furthermore, network usage statistics can be plotted for specific interfaces or accumulated across all interfaces of a node. The metrics include incoming and outgoing bytes per seconds, as well as total number of bytes per second 6 and number of packets and errors.

Another common metric that is often relevant is the number of contacts made between the nodes. These are the possible opportunities for data exchange in DTNs and emergency communication systems. We currently support visualization by

[8]https://github.com/adulau/Forban/
[9]https://github.com/gh0st42/goforban/

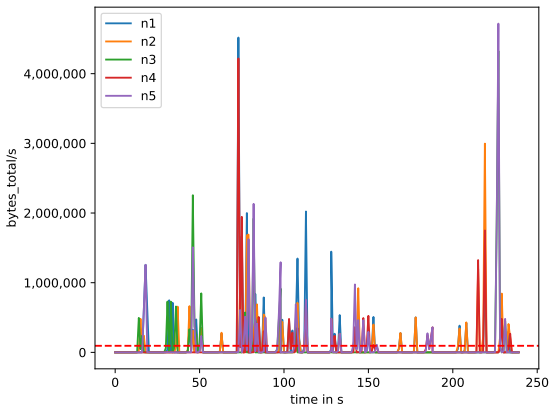


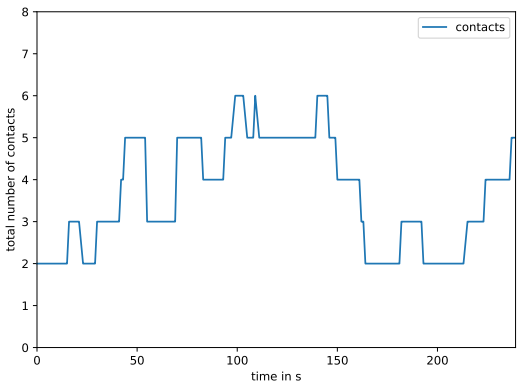
Fig. 6: Total bandwidth used per node over the experiment runtime. The dashed red line is the average across all nodes.

contacts over time (Fig. 7a) or the number of total contacts per node (Fig. 7b). The contacts as well as the network usage are highly dependant on the movement pattern chosen as well as the radio link properties, especially the simulated radio range and the available bandwidth.

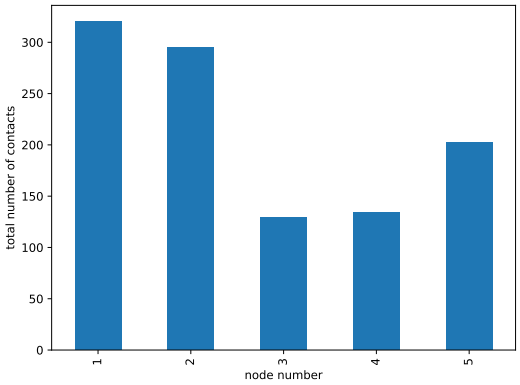
Overall, *coreemu-lab* needed 4:30min to complete the whole simulation, including the emulator setup, starting all monitoring services and doing the post experiment analyses. As the simulation runtime was set to 4 minutes, we have a 30s overhead for our five node setup.

VI. Conclusion

In this paper, we presented *coreemu-lab*, a novel framework for automated simulation and evaluation of networked apps. It can be used as an interactive tool to explore various challenging networking conditions with different software as well as headless operation for fully automated tests. Our approach gives one the ability to focus on the actual problems such as providing new communication systems for disaster response instead of learning new complex tools and repeating the same steps over and over again for evaluations. It is multi-platform and was tested on Linux and macOS, where, thanks



(a) Contacts over time.



(b) Total number of contacts per node.

Fig. 7: Analysis of main process statistics gathered in the example simulation for each node as averages and standard deviation. The dashed red line is the average across all nodes.

to *Docker*, it runs without further dependencies in a self-contained environment. The presented system is independent of an app language and works with anything from native binaries over Java bytecode to interpreted programs written in Python or node.js. We provide services to monitor standard metrics and later on turn these into plots and reports.

In the future, we want to incorporate even more monitoring and reporting specifically for DTN scenarios that should work with most existing disaster communication software out-of-the-box. Furthermore, we plan to integrate *coreemu-lab* with Software-in-the-Loop (SITL) simulations of the physical world to, e.g., see the effects of various radio-link and transport layer optimizations on the control of Unmanned-Aerial-Vehicles (UAVs). Finally, we hope to build a community around *coreemu-lab* that also contributes new features, scenarios and analyzers for many different use-cases.

ACKNOWLEDGMENT

This work has been co-funded by the LOEWE initiative (Hessen, Germany) within the *emergenCITY* center, as well as the German Research Foundation (DFG) in the Collaborative Research Center (SFB) 1053 MAKI.

REFERENCES

[1] A. Keränen, J. Ott, and T. Kärkkäinen, "The one simulator for dtn protocol evaluation," in *International Conference on Simulation Tools and Techniques (SIMUtools)*, 2009, pp. 1–10.

[2] A. Varga and R. Hornig, "An overview of the omnet++ simulation environment," in *International Conference on Simulation Tools and Techniques (SIMUtools)*, 2008.

[3] M. Fauzan, T. W. Purboyo, and C. Setianingsih, "Ibr-dtn to solve communication problem on post-disaster rescue mission," in *International Seminar on Intelligent Technology and Its Applications (ISITIA)*, 2019, pp. 24–28.

[4] L. Baumgärtner, P. Gardner-Stephen, P. Graubner, J. Lakeman, J. Höchst, P. Lampe, N. Schmidt, S. Schulz, A. Sterz, and B. Freisleben, "An experimental evaluation of delay-tolerant networking with serval," in *IEEE Global Humanitarian Technology Conference (GHTC)*, 2016, pp. 70–79.

[5] A. Sterz, L. Baumgärtner, R. Mogk, M. Mezini, and B. Freisleben, "Dtn-rpc: Remote procedure calls for disruption-tolerant networking," in *IFIP Networking Conference (IFIP Networking)*, 2017, pp. 1–9.

[6] R. Beuran, S. Miwa, and Y. Shinoda, "Performance evaluation of dtn implementations on a large-scale network emulation testbed," in *ACM International Workshop on Challenged Networks (CHANTS)*, 2012, p. 39–42.

[7] J. Ahrenholz, C. Danilov, T. R. Henderson, and J. H. Kim, "Core: A real-time network emulator," in *IEEE Military Communications Conference (MILCOM)*, 2008, pp. 1–7.

[8] S. Schildt, J. Morgenroth, W.-B. Pöttner, and L. Wolf, "Ibr-dtn: A lightweight, modular and highly portable bundle protocol implementation," *Electronic Communications of the EASST*, vol. 37, 2011.

[9] L. Baumgärtner, J. Höchst, and T. Meuser, "B-dtn7: Browser-based disruption-tolerant networking via bundle protocol 7," in *International Conference on Information and Communication Technologies for Disaster Management (ICT-DM)*, 2019, pp. 1–8.

[10] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," in *Modeling and tools for network simulation*. Springer, 2010, pp. 15–34.

[11] R. d. R. Fontes and C. E. Rothenberg, "Mininet-wifi: A platform for hybrid physical-virtual software-defined wireless networking research," in *ACM SIGCOMM Conference*, 2016, pp. 607–608.

[12] B. Richerzhagen, D. Stingl, J. Rückert, and R. Steinmetz, "Simonstrator: Simulation and prototyping platform for distributed mobile applications," in *International Conference on Simulation Tools and Techniques (SIMUtools)*, 2015.

[13] M. Stute, M. Maass, T. Schons, and M. Hollick, "Reverse engineering human mobility in large-scale natural disasters," in *ACM International Conference on Modelling, Analysis and Simulation of Wireless and Mobile Systems*, 2017, pp. 219–226.

[14] M. W. Kang and Y. Won Chung, "Performance analysis of a novel dtn routing protocol for icn in disaster environments," in *International Conference on Information and Communication Technology Convergence (ICTC)*, 2018, pp. 1276–1278.

[15] N. Schmidt, L. Baumgärtner, P. Lampe, K. Geihs, and B. Freisleben, "Miniworld: Resource-aware distributed network emulation via full virtualization," in *IEEE Symposium on Computers and Communications (ISCC)*, 2017, pp. 818–825.

[16] R. Beuran, L. T. Nguyen, T. Miyachi, J. Nakata, K.-i. Chinen, Y. Tan, and Y. Shinoda, "Qomb: A wireless network emulation testbed," in *IEEE Global Telecommunications Conference (GLOBECOM)*, 2009, pp. 1–6.

[17] R. Beuran, S. Miwa, and Y. Shinoda, "Performance evaluation of dtn implementations on a large-scale network emulation testbed," in *ACM International Workshop on Challenged Networks (CHANTS)*, 2012, pp. 39–42.

[18] J. Ahrenholz, T. Goff, and B. Adamson, "Integration of the core and emane network emulators," in *Military Communications Conference (MILCOM)*. IEEE, 2011, pp. 1870–1875.

[19] A. Froemmgen, D. Stohr, B. Koldehofe, and A. Rizk, "Don't Repeat Yourself: Seamless Execution and Analysis of Extensive Network Experiments," in *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2018.

[20] N. Aschenbruck, R. Ernst, E. Gerhards-Padilla, and M. Schwamborn, "Bonnmotion: a mobility scenario generation and analysis tool," in *International Conference on Simulation Tools and Techniques (SIMUtools)*, 2010, pp. 1–10.