Hardware-in-the-loop Simulation for Internet of Things Scenarios

Johannes Kölsch, Christopher Heinz, Sebastian Schumb, Christoph Grimm
TU Kaiserslautern
Erwin-Schrödinger-Straße 1, 67663 Kaiserslautern, Germany
Email: koelsch|heinz|s_schumb10|grimm@cs.uni-kl.de

***Abstract*—The Internet of Things is an increasingly complex ecosystem, which includes many different tightly interwoven domains. Therefore, the development of IoT applications and devices has to consider interoperability between various communication systems, domains, platforms and protocols. Throughout the whole development cycle continuous testing and validation can help to identify and conquer system errors. To facilitate this process, simulation is a suitable tool.**

In this paper, we present an approach for simulation of large-scale Internet of Things scenarios with Hardware-in-the-loop integration. This was achieved by extending omnet++ to connect a simulated model to real world devices.

***Index Terms*—Internet of Things, Simulation, Hardware-in-the-loop**

I. Introduction

Todays embedded and cyber-physical systems include parts from different, tightly interwoven domains, such as software, hardware, and mechanical parts. The increasing complexity and the multidisciplinary nature of these systems demands appropriate abstraction semantics to enable system design.

The complexity of developing cyber-physical systems is addressed by model-based design. Even if a fully implemented prototype is not available, it is possible to evaluate a virtual prototype of the system in model-based design. This leads to the possibility to reduce risks in an early system design phase considerably and enable reuse of already evaluated or tested models in later projects and products.[1] In cyber-physical systems, which tend to be very large and complex, these advantages of model-based systems engineering are improving the fundamentals of the development process.[2]

The development of cyber-physical systems is a task, that usually involves a lot of engineers, system architects and other developers from different domains and disciplines. All these stakeholders have to cooperate and communicate in a fast and efficient way. For these communication channels the old development process uses informal and mostly text-based documents, which due to freedom of interpretation is prone to errors. Models are replacing these documents more and more. This leads to a new problem: Different domains and different modeling levels also need different modeling languages and techniques.

At high system level the language SysML is becoming more and more popular to describe structure, behaviour and requirements of a complex system in a formal and understandable way.[3] On lower system levels it is of great importance, to describe components and their relations among each other in a very detailed way. Therefore it is indispensable to use domain-specific languages. For mechanical systems Modelica[4] is a popular language and for electrical systems on a high level SystemC/SystemC-AMS[5] is widely used.

A. The VICINITY project

VICINITY is an EU funded project under Horizon 2020.

The project started on the 1st of January 2016 and will last 4 years. The lack of interoperability is considered as the most important barrier to achieve the global integration of IoT ecosystems across borders of different disciplines, vendors and standards. Indeed, the current IoT landscape consists of a large set of isolated islands that do not constitute a real internet, preventing the exploitation of the huge potential expected by ICT visionaries.[6]

To overcome this situation, VICINITY presents a virtual neighborhood concept, which is a decentralized, bottom-up and cross-domain approach that resembles a social network, where users can configure their set ups, integrate standards according to the services they want to use and fully control their desired level of privacy. VICINITY then automatically creates technical interoperability up to the semantic level. This allows users without technical background to get connected to the vicinity ecosystem in an easy and open way, fulfilling the consumers needs. Furthermore, the combination of services from different domains together with privacy-respectful user-defined share of information, enables synergies among services from those domains and opens the door to a new market of domain-crossing services.

VICINITY's approach will be demonstrated by a large-scale demonstration connecting 8 facilities in 7 different countries. The demonstration covers various domains including energy, building automation, health and transport. VICINITY's potential to create new, domain-crossing services will be demonstrated by value added services such as micro-trading of demand side management capabilities, AI-driven optimization of smart urban districts and business intelligence over IoT. Open calls are envisioned in the project to integrate further, preferably public, IoT infrastructures and to deploy additional added value services. This will not only extend the scale of VICINITY demonstration, but also efficiently raise the awareness of industrial communities of VICINITY and its capabilities. [7]

For the purpose of demonstration, this work focuses on one of the four VICINITY pilots. However the described approach is not limited to this use-case but instead can be applied rather generic.

II. Proposed Approach

The TU Kaiserslautern is setting up a virtual environment to simulate, emulate and validate the platform and the semantic model in real life scenarios. The virtual environment is built up with the simulation environment omnet++. In the simulated environment like this, any use case of any size (even with thousands of IoT devices) can be built and directly tested with the real hardware integrated. This approach enables completely new possibilities in terms of testing and validation in early design phases. New devices or prototypes can be tested regarding their behaviour as a part of a large network and in different abstract scenarios. Also, a comparison between simulation results and actual implementations can be of great benefit. It is planned to simulate the whole VICINITY use case for the Energy Ecosystem of Martim Longo[8] in the virtual environment. For this purpose we model it in SysML. A direct automatic transformation of the SysML models into a mixed virtual and physical environment is under development and will be finished by end of this year.

A lot of challenges have to be conquered in the context of the developed simulation framework. One of the main challenges was to integrate omnet++ with real hardware, to enable a simulation with hardware in the loop. In this way, we can already test big Internet of Things scenarios, even if we only have a few sensors and actuators already available. In the following chapter, we describe how this problem was solved.

III. Hardware In The Loop

The process of inserting real hardware in otherwise simulated networks is often referred to as hardware in the loop simulation. Hardware in the loop simulations are especially useful in an Internet of Things context, as creating and monitoring test setups fully consisting of real hardware quickly becomes too complex. It is often more feasible to set up a network of simulated devices and connect to a small number of real devices. An additional advantage of this approach is that the internal state of the simulated devices and network components can be monitored, saved and restored easily. It also enables developers to compare the behavior of the real hardware, with simulations used in earlier stages of the development process.

In the following two different approaches for connecting real IoT devices to networks simulated using Omnet++ and the INET framework will be presented. In both cases the simulated network consists of a small number of virtual hosts communicating with each other, while three Raspberry Pis act as physical sensor nodes that communicate with the simulated hosts.

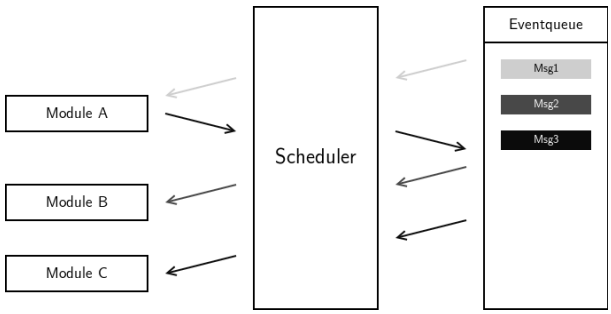

Fig. 1. Event scheduling in Omnet++

A. Omnet++

As shown in figure 1, Omnet++ bases its simulations on a central event queue. The event data structure consists of a source, a destination, a reference to a payload and an arrival time, which is used to simulate transmission delays between modules. Entries in the event queue are sorted in ascending order by their arrival time. The simulation scheduler [9], which is usually run in a single thread, retrieves the next event from the queue and waits for the difference between the current simulation time and the event's arrival time. As soon the simulation time has caught up with the event's arrival time, the scheduler passes the event on to its destination module, by calling the module's event handler function. The event handler function can then generate new events in response, and insert them into the event queue by returning them to scheduler. When control flow resumes in the scheduler, it continues with the next event in the queue. In most cases simulation time is decoupled from wall clock times, which means waiting for event arrival times can be trivially implemented by incrementing the simulation time.

In order to perform hardware in the loop simulations, the simulation has to be run at realtime speed and the simulation time need to be synchronized with wall clock time. Furthermore it is necessary to insert packets from the host's network interfaces into the simulations event queue. As the control flow is only handed over to a module, to process an event for the module, it is not possible to capture packets from the host systems network card directly inside a module. Instead, the scheduler has to be modified to monitor network cards and to handle incoming packets, while it waits until the next event can be processed. Even though it is possible to use the serializers and deserializers provided by the INET framework to convert packets from the real network into events for the simulation and vice versa, the deserializers can only invoked from inside a module's event handler function, as they will attempt to use a reference to the calling module as the event's source module. Therefore, they can not be used directly by the scheduler. To work around this limitation, the raw bytes making up the packet can wrapped inside an event, which is then added to the event queue to be send to a simulation module. The module can than use a deserializer to convert the bytes into an event containing a packet for the simulated

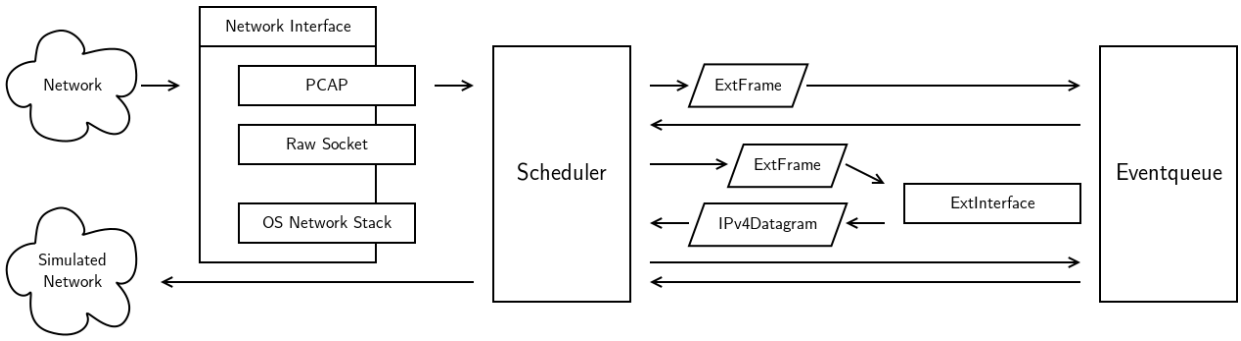

Fig. 2. Capturing a packet addressed to the simulated network

network. The arrival time for these events has to be chosen carefully to preserve the overall order of packets across the two networks. Both of the schedulers presented here assume, that the simulation runs with real time speed, so that the scheduler can use the current simulation time as arrival time for the packets from the real network. If the simulation runs slower than real time the packets could be stored until the simulation has caught up, but the devices in the real network are unaware of the slower time inside the simulation and will likely run into timeouts. The reverse direction, forwarding packets from the simulated network to the real network, is simple: Modules can use a serializer to turn an event from the simulated network into a packet for the real network. The packet can then be handed over to the scheduler, which can forward it into the real network.

The routed approach (presented in III-B) uses modules already included the INET framework, to forward IP traffic [10] to and from a physical network interface of the host computer running the simulation into the simulated network. As this technique only works on IP traffic, it is independent from the underlying network layers used in the real network, but it also limits the simulated network to routers and hosts interconnected with point-to-point IP links. Devices in the real network have to be configured to use to the host computer as gateway for the IP subnets of the simulated network. Depending on the device used it might not always be possible to set up static routes, other than the default gateway.

The switched approach (presented in III-C) works on the ethernet layer and requires additional custom simulation modules, as well as a custom scheduler. It allows connecting multiple wired network interfaces of the host system, to the simulated network. In contrast to the previous approach the simulated network can have an arbitrary topology, and can use any link layer protocol, as long as the modules interfacing with the real network are ethernet based. This implementation is intended to be used primarily on linux systems, as it relies heavily on the tap interfaces and network bridges provided by the linux kernel.

B. Routed Approach

This setup simulates a routed network consisting only of IP point-to-point links. It is based on the INET framework's extserver example. The example uses the ExtInterface module and the cSocketRTScheduler class already included in the framework's examples.

As shown in figure 2 the scheduler uses the PCAP library [11], which is also used by network monitoring tools like e.g. tcpdump or Wireshark, to capture IP packets from one of the host computer's network interfaces, while it waits for the arrival time of the next event. It sets up a PCAP filter to restrict the captured packets to those addressed to subnets inside the simulated networks. If a packet has been captured the scheduler interrupts waiting for the next event, wraps the IP packet in an ExtFrame message, which has a ExtInterface module as destination, and inserts the message into the event queue. The current simulation time is used as arrival time. Since it is assumed that the simulations runs at a speed close to real time, this should by definition preserve correct ordering of events inside the simulation. After the new event was added, the scheduler has to check whether, the event it was initially waiting for, still is the earliest event in the queue. In most cases the newly added ExtFrame will be the earliest event, as it has been added using current simulation time as arrival time. The scheduler will then send the ExtFrame to the ExtInterface module, by passing it to the module's message handler method. The handler method then uses a deserializer to create an IPv4Datagram message, which it then sends into the simulated network via its point-to-point link.

Figure 3 shows how IP packets routed to the ExtInterface, having destination addresses from a subnet in the real network are serialized and handed over to the ExtInterface module. The ExtInterface module uses a raw socket on the host system's network interface to send the packet into the real network.

PCAP and raw sockets can be used on any platform, as they do not require any special features from the host systems network stack. Therefore this approach is highly platform independent. Additionally, PCAP's integrated filter feature also provides an efficient way to efficiently filter out traffic that is intended for the simulated network. However, the major drawback of this approach is that does not allow arbitrary network structures inside the simulation and is limited to IPv4 based unicast traffic. As a result the protocols relying on multicast or broadcast packets for device discovery can not be used. Furthermore, ARP [12] requests and responses

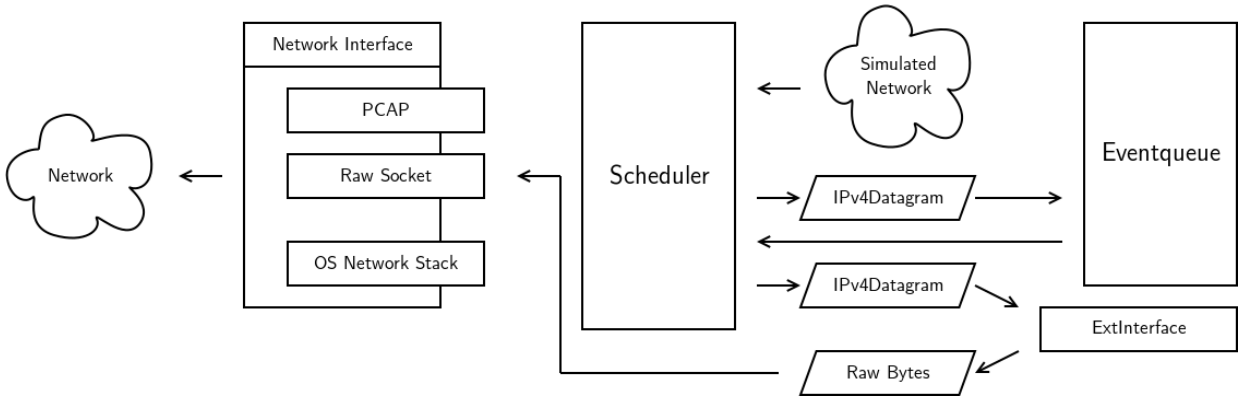

Fig. 3. Forwarding a packet from the simulation to the real network

to and from the simulated network are not forwarded, which means that devices in the real network can not use ARP to determine to which hardware addresses packets for hosts inside the simulated network should be sent. Instead, it requires the additional configuration of static routes to the simulated network, using the host system as gateway, on each device in the real network. Especially on simpler low-cost IoT devices these configurations options might not be easily accessible. Additional the scheduler and the interface module are currently only able to process IPv4 packages, which poses a problem as IPv6 [13] and the IPv6-based W6LoPan [14] become increasingly important for IoT-applications.

C. Switched Approach

The switched network simulation has been designed specifically to avoid the limitations exhibited by the extserver example.

Instead of PCAP it utilizes the linux kernel's build in tap interface [15] and network bridge support. Towards the kernels network stack tap interfaces behave exactly like any other network interfaces does. This means they can be used in more advanced network setups involving routing, firewalls, traffic shaping and network bridges. In contrast to regular network interfaces the ethernet frames generated for them by the network stack are not passed on to a physical network card. Instead, they are forwarded to special buffer, which can be accessed by userspace programs via a special device file. This enables the program to read any raw ethernet frames which are received by the tap interface and to send out frames by writing them to the buffer. Tools like OpenVPN usually use tap interfaces to connect a layer-2 virtual private network to the kernel's network stack. Network bridges [16] can be used to connect two or more network interfaces of a linux system with each other as if they were ports on a virtual switch. This means if a packet is received by one interface it will be sent out again on the other interface, if the bridge's mac address table indicates that the destination can be reached from the other interface. Additional the kernel creates a bridge interface for the bridge which acts as an additional virtual network card, connected to the virtual switch.

Similar to the routed approach presented previously, the tap interface can not be read directly by an Omnet++ module, as the control flow only enters a module for a short period of time in order to process an event from the event queue. Therefore, it is not possible to use blocking reads on the tap interface. On the other hand it is also not feasible to use non-blocking reads in side a module, as it is not guaranteed that the scheduler will call the handler code in the module on a regular basis. This is solved by moving the actual input and output code into the scheduler, while using module for serialization and deserialization, as the serializers and deserializers can only operate from within a module context. As shown in figure 4 this implemented using two custom modules: TapRTScheduler and TapInterface. Additionally, the EtherTap module is required to wrap the TapInterface module and add an Ethernet-MAC module provided by INET, that converts frames to and from the physical layer encoding, required for INET's Ethernet simulation.

The TapInterface modules provide interface parameter, specifying the name of the tap interface the module is connected to. During module initialization of the simulation, the scheduler initializes all tap interfaces and stores them in a hashmap mapping the TapInterface modules to the file descriptors of the corresponding tap interfaces.

Each scheduling cycle starts with TapRTScheduler reading the next event from the event queue and using the arrival time of the event to determine if and how long it should delay handing the event over to the receiving module. It is important to keep in mind that the scheduler calculates current simulation time as wall clock time that has passed since the simulation has been started or resumed, as it assumes that the simulation runs with realtime speed. If there is no next event, the scheduler attempts to delay for LONG_MAX seconds. In case the difference between simulation time and the event's arrival time is zero or negative the scheduler directly proceeds to call the receiving modules handler method. For positive differences the TapRTScheduler has to delay until the simulation time has reached the event's arrival time. The actual delay is implemented using the select-syscall to wait until either the required amount of time has passed or a packet

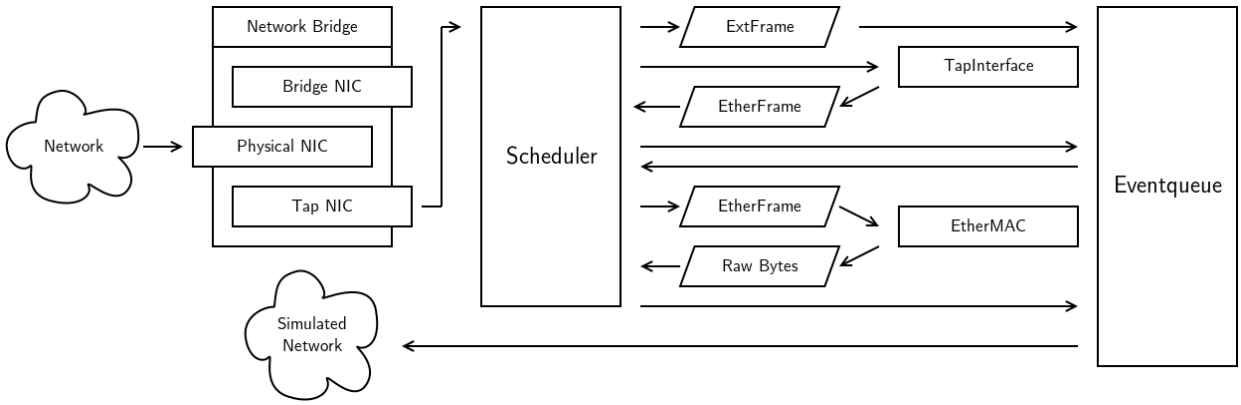

Fig. 4. Forwarding a packet into the simulation

has arrived on one the tap interfaces. Additionally the method is required to update the user interface periodically during the delay. This mechanism interleaves waiting for the next event, reading packets from the tap interfaces and reacting to user interface events.

Any received Ethernet frame is passed to the corresponding TapInterface module for deserialization. Similar to the cSocketRTScheduler the raw bytes, representing a full Ethernet frame, are wrapped inside an ExtFrame message and enqueued in the event queue, as the deserializer can only be used from inside a module's message handling method. The ExtFrame is created using the current simulation time as arrival time and the TapInterface module associated with the file descriptor from which the frame was read as destination.

As soon as the control flow returns from the delay method to the actual scheduler, there are two possible cases to consider: The first case is that there were no Ethernet frames available at any file descriptor, and therefore the entire difference to the event's arrival time has been spent waiting for data. In this case the scheduler can directly pass the event to the receiving modules message handling method. In the second case it is too early to hand the event over to the receiving module and implying that at least one Ethernet frame was read and one new ExtFrame event, with an earlier arrival time has been added to the queue. The arrival time will always be earlier for these events, as the current simulation time has been used to create the ExtFrame, while waiting for the previously selected event to arrive at its destination module. Therefore, the scheduler has to enqueue the initially selected event again and dequeue the earliest ExtFrame. In both cases the scheduler can pass the event to its destination module, for processing and the next scheduling cycle starts as soon as the control flow returns from the modules message handler method.

When a TapInterface receives a ExtFrame, it unpacks the raw Ethernet frame inside and uses a EthernetSerializer to deserialize it into an EtherFrame instance. After that the EtherMAC that is part of the EtherTap module wrapping the TapInterface module encapsulates the frame in an EtherPhyFrame, which represents Ethernet physical layer encoding and sends it into the simulated network.

Forwarding packets from the simulated network to the real network is a much simpler process. As shown in figure 5 any EtherPhyFrame received by a EtherTap module is decapsulated by the modules EtherMAC- The TapInterface then serializes the EtherFrame into raw byte representing an ethernet frame, which can to be written directly to a tap interface. Next the scheduler writes the frame to the file descriptor associated with the module. This results in the Ethernet frame being send out of the tap network interface and therefore out of any physical network card connected to the same bridge interface as the tap. It is important to keep in mind at this point, that since the bridge interface acts as a transparent switch, it is not necessary to change the mac addresses in the Ethernet frame. The simulated and the real network form a single broadcast domain.

This implementation limits the platform compatibility to linux based systems. Windows and MacOS do not offer tap interfaces and while it is possible to add them using third party software, the API for accessing the interface is different and would therefore require platform dependent adjustments to the scheduler. Additionally the network bridging may not be available using these third party solutions.

An other disadvantage using tap interfaces is that the network has to be simulated down to ethernet level, which requires significantly more resources than just simulating IP point to point links. On the other hand it may be desirable to be able to simulate arbitrary network topologies, or protocols not based on-top of the IP-protocol, which would require a switch to an ethernet level simulation anyway.

The major advantage of this approach is that it combines the simulated and the real network into a single broadcast domain. This does not only enable arbitrarily structured IP networks on top of it, but also eliminates the need for setting static routes into the simulated network on the real devices. Additionally, it also enables the usage of broadcast and multicast based discovery protocols.

Unlike in the PCAP based approach, the network packets transmitted through the tap interface are actually processed by the host systems network stack, which allows making use of wide variety of tools offered by the linux network

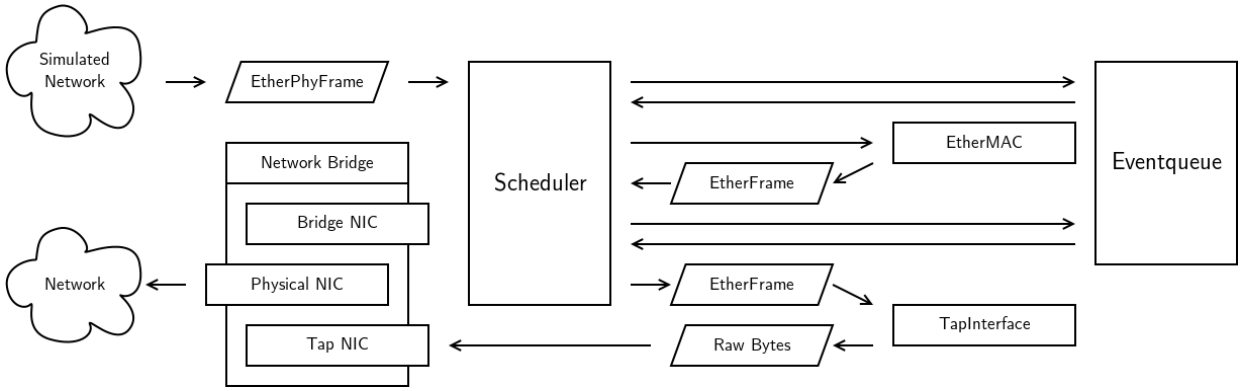

Fig. 5. Forwarding a packet to the real network

ecosystem. For example, it is possible to setup sophisticated firewalls using iptables to limit the communication between different the simulated and the real network. It also possible to use traffic shaping tools such as tc to restrict the bandwidth between the simulated on the real network or implement a basic traffic shaping. Furthermore, the same tools that are commonly used for identifying network problems, such as ping, traceroute, or tcpdump can now be applied to find problems in the simulated network as well.

IV. Conclusion and Outlook

In this paper we demonstrated how real hardware can be integrated into an omnet++-simulation environment. A demonstrator with two real sensors and a virtual actuator has been built to show how the simulation can interact with the real world.

For performance evaluation a large scale scenario has to be completed and a suitable metric has to be developed. This is currently in progress.

Using standard interfaces provided by the linux network stack enables us to make use of wide variety of existing tools in the linux networking ecosystem, such as firewalls, traffic-shapers, QoS-implementations and tools for traffic analysis. Additionally network bridges can also be used as a layer 2 connection between multiple Omnet++ simulations allowing us to simulate different sections of a larger network in parallel, without having to implement a dedicated Omnet++ scheduler for parallel or distributed simulation.

The vision is to integrate this approach into a platform for modeling and simulation of big Internet of Things scenarios, like a complete VICINITY pilot site with thousands of nodes, gateways, sensors and actuators. This platform is currently under development. It enables parallel and distributed simulation in an hierarchical way. On the low level simulation side, we are integrating domain-specific languages, like Modelica and SystemC-AMS, for finer simulation results over FMI-Interfaces.[17]

Furthermore we are currently developing a tool to close the gap between the high-level systems modeling language SysML[3] and an executable simulation model, which can be generated and simulated very fast and easy in an early development phase. Thus, errors in the system design can be identified and conquered in an earlier design phase.

Acknowledgement

As a part of the VICINITY project, this work has been supported by EU (European Union) program Horizon 2020 under grant agreement number 688467.

References

[1] P. Derler, E. A. Lee, and A. S. Vincentelli, "Modeling cyber–physical systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 13–28, 2012.
[2] D. Gianni, A. D'Ambrogio, and A. Tolk, *Modeling and Simulation-Based Systems Engineering Handbook*. CRC Press, 2014.
[3] Object management group, specification of sysml 1.4. [Online]. Available: http://www.omg.org/spec/SysML/1.4/PDF
[4] M. Association. The modelica language. [Online]. Available: https://www.modelica.org/
[5] Accelera systems initiative, systemc. [Online]. Available: http://www.accellera.org/downloads/standards/systemc
[6] A. Mynzhasova, C. Radojicic, C. Heinz, J. Kölsch, C. Grimm, J. Rico, K. Dickerson, R. Garcıa-Castro, and V. Oravec, "Drivers, standards and platforms for the iot: Towards a digital vicinity."
[7] Vicinity project website. [Online]. Available: http://vicinity2020.eu/
[8] Martim longo (portugal) - neighbourhood grid ecosystem. [Online]. Available: http://vicinity2020.eu/vicinity/content/martim-longo-po-neighbourhood-grid-ecosystem
[9] Omnet++ library: cscheduler class reference. [Online]. Available: https://www.omnetpp.org/doc/omnetpp/api/classomnetpp_1_1cScheduler.html
[10] Internetwork protocol specification. [Online]. Available: https://www.rfc-editor.org/ien/ien54.pdf
[11] Manpage of pcap. [Online]. Available: http://www.tcpdump.org/manpages/pcap.3pcap.html
[12] An ethernet address resolution protocol. [Online]. Available: https://tools.ietf.org/html/rfc826
[13] Internet protocol, version 6 (ipv6) specification. [Online]. Available: https://tools.ietf.org/html/rfc2460
[14] Transmission of ipv6 packets over ieee 802.15.4 networks. [Online]. Available: https://tools.ietf.org/html/rfc4944
[15] Universal tun/tap device driver. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/tuntap.txt
[16] Linux network bridges. [Online]. Available: https://wiki.linuxfoundation.org/networking/bridge
[17] Modelica association project, the fmi standard. [Online]. Available: https://www.fmi-standard.org/