EPiC Series in Computing

Volume 56, 2018, Pages 1–10

Proceedings of the 5th International OMNeT++ Community Summit

jUDPWrapper: A Lightweight Approach to Access the OMNeT++/INET UDP Functionality from Java

Henning Puttnies[1], Peter Danielis[1], Leonard Thiele[1], and Dirk Timmermann[1]

University of Rostock, Faculty of Computer Science and Electrical Engineering, Institute of Applied Microelectronics and Computer Engineering, Germany
henning.puttnies@uni-rostock.de

Abstract

There are three methods to evaluate networked systems during the design phase: by means of simulation, a practical testbed, and mathematical analysis. In this paper, we propose the jUDPWrapper, which enables to access the OMNeT++/INET UDP functionality from Java using the Java Extensions for OMNeT++. The jUDPWrapper provides a socket-based API corresponding to the standard API defined in the java.net package. Consequently, it enables to develop application layer simulation models in Java and hereafter easily derive a Java prototype implementation. Therefore, the jUDPWrapper eases the evaluation of Java UDP applications using both simulation and practical testbed. We evaluate the jUDPWrapper using two distinct versions of OMNeT++/INET, provide the Java Extensions for OMNeT++ 5.4, and for the first time quantify the execution time of Java models versus C++ models. C++ models are approx. twice as fast as Java models in long simulation runs, if the release mode is used.

1 Introduction

The evaluation of networks during the design phase is possible using three different methods, namely: simulation, testbed, and math analysis [11]. Network simulation frameworks (e.g., the Objective Modular Network Testbed in C++, OMNeT++) are specifically important tools to evaluate new approaches (e.g., for security, communication, time synchronization) in large scale scenarios with many networked nodes, as a testbed implementation is both expensive and time consuming. Widely used simulation frameworks like OMNeT++ provide the ability to reuse existing modules. This allows to setup a simulation with comparably low effort.

On the other hand, the Java programming language is suitable for rapid prototyping in practical testbeds as it is relatively easy to debug and platform independent. Therefore, we extensively use Java applications in our testbeds (e.g., in the Mini-Mesh testbed [8] or for the evaluation of the PSPI-Sync approach [4]). Moreover, Java is even suitable for real-time applications in the Industrial Internet of Things (IIoT) as shown in [6].

Our idea of an enhanced evaluation methodology is to first develop Java simulation models using OMNeT++ and afterwards derive a Java prototype implementation. Finally, the real world results (e.g., delays, energy consumption, background traffic) can be fed back into the

A. Förster, A. Udugama, A. Virdis and G. Nardini (eds.), OMNeT 2018 (EPiC Series in Computing, vol. 56), pp. 1–10

simulation to get more precise results for large scale scenarios. Although this is already possible for C++ models, using Java and OMNeT++ can be seen as a trade-off between implementation effort and computation time.

As the Java Extensions for OMNeT++ (JEO) exist [9], the next logical step is to provide a socket-based API for the OMNeT++/INET UDP functionality to ease the implementation of Java application layer simulation models (ALSM) as well as the derivation of prototype implementations. This approach is similar to the classes *DatagramSocket* and *UDPSocket* for C++ ALSMs and the API is consistent with the Java standard API in the *java.net* package. We focus on UDP in this paper, as many IoT protocols run over UDP (e.g., CoAP, MQTT-SN) and since it is more lightweight than TCP. Our contributions are as follows:

- Development of the jUDPWrapper, which provides a simple and socket-based interface to INET's UDP functionality.
- The jUDPWrapper also serves as an example of how to access INET modules from Java, as we describe a generic approach to access message fields that have a custom data type (e.g., an instance of a C++ class). Applications of this are the *UDPSendCommands* of INET that comprise source and destination addresses being of the custom type *L3Address*. This is important for the jUDPWrapper as we have to communicate with the *udp* module of INET. Moreover, this is important for the JEO in general as the interface towards INET is crucial to reuse existing modules. This was also a major challenge for the development of the jUDPWrapper as the JEO supported only the access to fields of standard types previously (e.g., int, double, and bool).
- We contribute with different example applications to show the interoperability between Java ALSMs and INET modules as well as C++ ALSMs and to ease the development of custom Java ALSMs. We evaluate the jUDPWrapper using both: OMNeT++ 5.0 with INET 3.4.0 and OMNeT++ 5.4 with INET 3.6.4. Thus, we also provide the Java Extensions for OMNeT++ 5.4. Concerning the performance, we demonstrate that C++ ALSMs are approx. twice as fast as Java ALSMs in long simulation runs and the release mode. However, this speedup is reduced in debug mode or if a mixed language setup is used (e.g., some C++ ALSMs and some Java ALSMs).
- Finally, the entire System (OMNeT++, JEO, jUDPWrapper, and examples) is publicly available [1]. These examples are easy to use as there is no configuration effort (e.g., for the Java class paths or linking towards several shared libraries). Moreover, everyone can retry the performance measurements.

The rest of this paper is organized as follows. Section 2 gives an overview of the basics of jUDPWrapper: the JEO and the UDP implementation in OMNeT++/INET. Section 3 describes related works and their suitability for Java ALSMs. Section 4 describes the conception of the jUDPWrapper and sketches its implementation. The evaluation is given in Section 5, and Section 6 concludes the paper.

2 Basics

Before describing the jUDPWrapper, we want to give some basic information about the JEO as well as the current implementation of UDP in INET in order to ease the understanding of

[1] We share the VM as *.ova:
https://unibox.uni-rostock.de/dl/fi63hr2MKwtk1RaysdMrzgnV/.zip

this paper.

2.1 Java Extensions for OMNeT++ (JEO)

The JEO base on Java Native Interface (JNI), were available from OMNeT++ 3.X to 4.6, and were newly generated for OMNeT++ 5.X [9]. The JEO provide a new simulation executable called *jsimple*. This executable comprises the OMNeT++ simulation kernel as well as several interface classes and methods. The class *jSimpleModule* allows implementing the functionality of simple modules in Java in the same manner as *SimpleModule* allows implementing in C++. Accessing fields of *cMessages* and *cObects* (or derived classes) is possible by using the *setField()* and *getField()* methods that implement the interface between the C++ and Java domain using strings as intermediate types.

2.2 The Implementation of UDP in OMNeT++/INET

The UDP protocol is implemented in a module named *udp* in INET. An important part is the class *UDPControlInfo*. An instance of this class is attached to every message that is sent from the ALSM to *udp*. The *UDPControlInfo* specifies whether the message contains: data and thus is a UDP datagram (*UDPDataIndication*) or it is a command message, e.g., to bind a socket (*UDPBindCommand*), connect a socket (*UDPConnectCommand*) or close a socket (*UDPCloseCommand*).

Furthermore, INET provides the *UDPSocket* convenience class to ease the use of the UDP functionality. This class provides a socket-based API with functions to create and configure a socket and send datagrams [2] (e.g., *bind()*, *connect()*, *send()*). Finally, INET provides the module interface *IUDPApp* for UDP apps. *IUDPApp* severs as template for ALSMs that are applicable in *StandardHost* [2].

3 Related Work

In this section, we refer to related works and analyze their support of Java simulation models.

The network ns-3 simulator (ns-3) [5] supports the execution of applications software using Linux containers (LXC) [1]. This also provides the ability to execute Java applications, if the container has installed a JVM. However, this approach has two main drawbacks. Firstly, this only allows to execute ALSM in contrast to the JEO were Java simulation models can reside at any network layer. Secondly, it is very difficult to synchronize the virtual times of the simulator and the guest system. One research approach to accomplish this is TimeKeeper [7] but there is the need to modify the Linux kernel accordingly and there is still an error between the actual simulation time and the time of the guest system.

Moreover, ns-3 provides the Direct Code Execution (DCE) to run existing implementations without source code changes, e.g., for using real implementations of userspace or kernelspace network protocols. DCE overrides the systems calls in order to create an interface between the implementation and the ns-3 simulator. Unfortunately, DCE only supports C++ source code.

The Fast Network Simulation Setup (FNSS) [10] is a tool chain that simplifies the process of setting up a network experiment scenario and supports various frameworks for network simulation and emulation like ns-2, ns-3, Mininet, Omnet++, Autonetkit and jFed. FNSS supports the Python, C++, and Java programming languages. However, the applied simulation framework restricts the tool. In other words, using Java simulation models is possible only if supported by the simulator.

As noted in [9], there are several Java compatible network simulation frameworks, that are not longer under maintenance: JNS, JNetworkSim, JProwler, Java Simulator, and SSFnet.

To conclude, there is, to the best of our knowledge, no other simulation framework that supports Java simulation models as good as OMNeT++ and the JEO.

4 jUDPWrapper

In this section, we describe the jUDPWrapper that uses the JEO. Hence, the jUDPWrapper serves also as example use of the JEO. Firstly, we highlight its components and a few of the most important API methods. Secondly, we propose a generic approach to access message fields that have a custom data type. The proposed approach is the successor of several considered ones. See section 4.3 for the alternatives. For the conception of the jUDPWrapper we state the following design targets:

I It should fit optimally into the existing OMNeT++/INET ecosystem. However, no modifications shall be made to the OMNeT++/INET code.

II It should be as lightweight as possible to easily keep track with progressing versions of OMNeT++/INET.

4.1 Access the OMNeT++/INET UDP Functionality from Java

As depicted in Fig. 1(a), the Java ALSM uses the jUDPWrapper to interface with INET. The jUDPWrapper translates the socket-based API calls into messages to the INET *udp* module. Writing a Java ALSM using the jUDPWrapper is similar to writing C++ ALSMs with INET. As depicted in Fig. 1(b) the Java ALSMs are compatible with INET's *StandardHost*. It has to implement the *IUDPApp* module interface defined in INET (ned: *like IUDPApp*) and it has to extend (ned: *inherits from*) the *JSimpleModule* to use the OMNeT++ simulation kernel in Java. Furthermore, we provide convenient classes to be used by the Java ALSM: *DatagramSocket* and *InetAddress* which provide the same API as *java.net.DatagramSocket* and *java.net.InetAddress*.

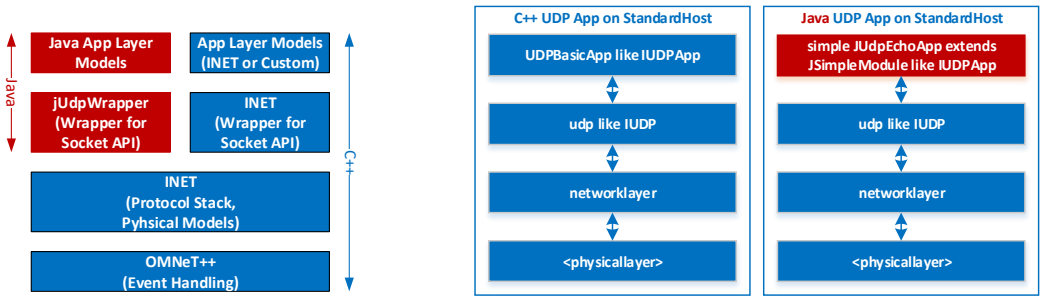
The *DatagramSocket* provides several standard methods that provide exactly the same API as defined in the java.net package, such as: *bind()* to bind a socket to a given IP address and port, *send()*: to send a UDP datagram to a remote host, *sendTo()*: to send a UDP datagram to a remote host that is determined by a given IP address and a port, and *close()*: to close the socket.

4.2 Accessing Message Fields of a Custom Data Type from Java: The L3Address as an Example and the General Approach

Generally, there is a problem when accessing field of a *cMessage* or *cObject* that are of a custom type (e.g., *L3Address*). The *getField()* method works correctly for all data types. However, the *setField()* method uses string casting for the handover of data from the Java to the C++ domain and only works for standard types (e.g., *int*, *double*, *bool*).

Therefore, an important development question to be answered is: how to access a field of a custom type? An example of message fields that have a custom data type are the destination and source IP addresses within a UDP datagram. Referring to INET, this fields are of type *L3Address* and part of, e.g., the *UDPSendCommand*.

Our approach to access such a field is using its string constructor. Therefore, we utilize a special syntax in the **.msg* file to automatically link the *setFied()* method to the corresponding



(a) (b)

Figure 1: (a) Basic structure of interface between Java ALSMs and INET (b) Comparison between *UDPBasicApp* (INET) and a Java ALSM using the jUDPWrapper.

string constructor. We highlight this as this syntax is only documented in the OMNeT++ source code. Although depending on undocumented features is usually discouraged, using this syntax in the **.msg* file has several benefits. We move the modified **.msg* files into the *jsimple* project. Therefore, this approach is completely independent of the INET project where the original **.msg* files still remain.

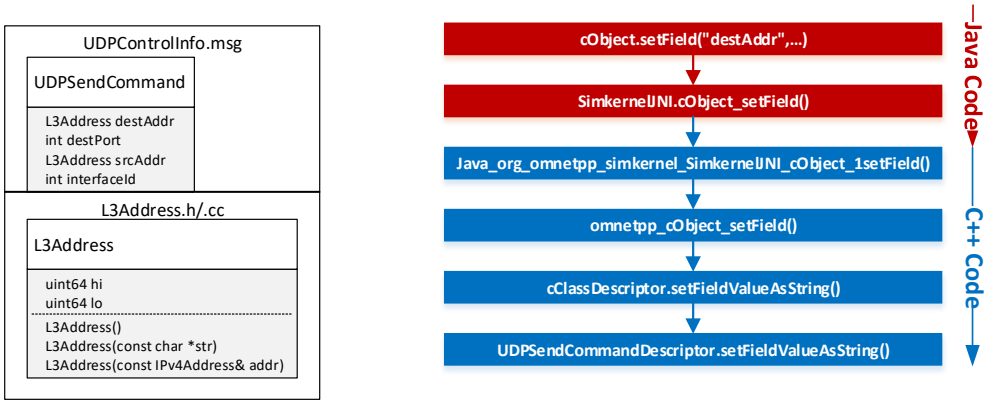
This approach has several benefits. It is lightweight as we do not change any source code of OMNeT++/INET. Generally, only a few modifications in **.msg* files are needed (one modification for every field). These changes lead to changes in the C++ code generated by the message compiler (MSGC). Consequently, this approach is easily applicable to other messages that comprise custom fields. Therefore we considered this approach to be of a general interest to interface between Java and INET. Moreover, the **.msg* file can also be changed in INET, if future changes shall not be done in two places.

To highlight the functionality we will describe the approach in an example. Fig. 2(a) depicts the simplified class diagrams of the *UDPSendCommand* and the *L3Address*. It is notable that the *UDPSendCommand* comprises the fields *destAddr* and *srcAddr* of the type *L3Address*, which has a string constructor. For the sake of comprehensibility, Fig. 2(b) states the entire call hierarchy. Note that a call of the Java method *cObject.setField()* finally calls the C++ method *UDPSendCommandDescriptor.setFieldValueAsString()*. Fig. 3 depicts the comparison between the original and the modified **.msg* files as well as the generated **.cc* files. Using the expression *@editable @fromstring(...)* in the **.msg* file, leads to access of the field *destAddr* in the generated C++ code. As a consequence, the *setField()* method has access to *destAddr* now.

4.3 Discussion of Alternative Approaches to Access Fields of a Custom Data Type from Java

The proposed jUDPWrapper approach demands changes only in the **.msg* files. As it is the successor of several considered approaches, we will discuss the alternatives shortly in the following.

The first alternative approach is writing a C++ interface module for the translation between string fields and fields of custom types. This approach has the benefits that it is independent of INET, if the C++ code remains inside the *jsimple* project. Furthermore, it is transparent for



(a) (b)

Figure 2: (a) UML model of *UDPControlInfo* and *L3Address* (b) Call hierarchy for setting the *destAddr* field of an *UDPSendCommand*.

INET modules as *L3Address* fields still exists in the **.msg* files. The drawback of this approach is its complexity. The application layer module would consist of both a Java ALSM and a C++ converter module. This would be less straightforward for the user than simply using the *IUDPApp* interface for Java ALSMs analogue to C++ ALSMs. Moreover, this approach would be less lightweight. Firstly, it needs a completely new conversion module for every INET module that needs to be interfaced. Secondly, there would be one "Java clone" for every **.msg* file that hat the same fields as the original **.msg* file but this fields have the type string (note that in our approach it is still possible to only modify the **.msg* file in INET). Finally, the adaption effort would be higher as the code of the C++ interface module must be changed, if the interface of INETs UDP functionality changes (e.g., if it uses a different type for the addresses than *L3Address*).

The second alternative approach is to add an additional string field in the **.msg* file and writing some custom setters that always set the *L3Address* field when the string field is set and vice versa (need to be done for every message between INET and Java). Again, this approach has the benefits that it is independent of INET, if C++ code is inside the *jsimple* project and it is transparent for the INET modules as the *L3Address* fields still exists. However, the drawbacks are that we need custom C++ setter methods for every message (in contrast, the current approach only demands changes the **.msg* file). Furthermore, there would be two fields holding the identical information. This would be less straightforward for the user than simply using the original *L3Address* fields in the Java code similar to C++ code.

The third approach is to modify the Java extensions so that *setField()* does not use *setStringAsValue()* but other setters. However, this approach has many drawbacks as it is less safe. If the field is not set as string, this might lead to run time errors as Java extensions base on JNI and dynamic linking (in contrast to compile time errors). Consequently, this approach still needs string casting for safe interpretation of data. Furthermore, it might need changes to the INET code, if there are no valid setters that can be used by *setField()*.

The fourth approach would be wrapping the entire INET library using SWIG. However, this

Listing 1: original **.msg*

```
class UDPSendCommand extends
    UDPControlInfo{
L3Address destAddr;
int destPort = -1;
//...
}
```

Listing 2: modified **.msg*

```
class UDPSendCommand extends
    UDPControlInfo{
L3Address destAddr @editable
    @fromstring(inet::L3Address($));
int destPort = -1;
//...
}
```

Listing 3: original **.cc*

```
bool UDPSendCommandDescriptor::
    setFieldValueAsString(/*...*/)
    const{
//...
switch (field) {
    case 1: pp->setDestPort(
        string2long(value)); return
        true;
    //...
}}
```

Listing 4: modified **.cc*

```
bool UDPSendCommandDescriptor::
    setFieldValueAsString(/*...*/)
    const{
//...
switch (field) {
    case 0: pp->setDestAddr(inet::
        L3Address(value)); return
        true;
    case 1: pp->setDestPort(
        string2long(value)); return
        true;
    //...
}}
```

Figure 3: Comparison between original and modified **.msg* files as well as the generated **.cc* files. Accessing the field *destAddr* is possible using the expression *@editable @fromstring(...)* in the **.msg* file.

has many drawbacks as many methods would be wrapped and made accessible from Java but no simulation setup would use the entire INET functionality. Moreover, this approach is very dependent on the INET version and needs high effort for the adaption to new INET versions. It demands changes in the SWIG interface files, and maybe the development of C++ wrapper functions (e.g. for C++ operator overloading that is not supported in Java). Another problem is that SWIG has issues with namespaces. E.g., two C++ methods *Foo1::Bar()* and *Foo2::Bar()* are wrapped into two similarly named Java methods *Bar()*. As *Bar()* is defined multiple times, the compile process crashes. This can be solved using SWIGs *%rename* directive, but it would lead to an unacceptable effort to do this for the entire INET library.

5 Examples and Evaluation

In the following, we will propose several example networks that show the interoperability between Java ALSMs, INET modules, and C++ ALSMs as well as evaluate their performance.

5.1 C++ INET UDP Example Network

The first example only consist of an Ethernet network and several *UDPBasic* ALSMs. We developed this example to show how to use *UDPBasic* of INET and how to setup a simple

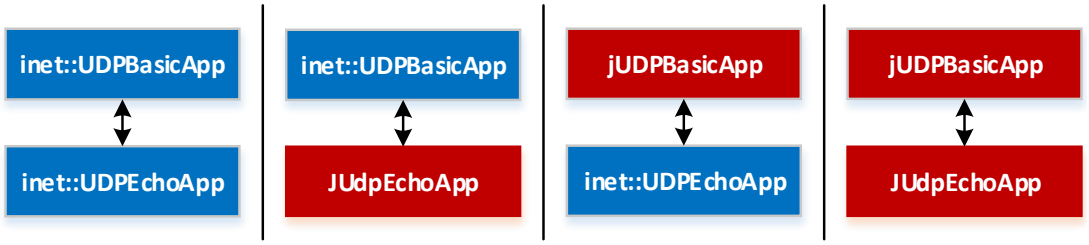


Figure 4: Four different examples used for evaluation of interoperability and performance of Java and C++ ALSMs. Note that every network comprises two *StandardHosts* that run the ALSMs and are connected via an Ethernet switch but this is not shown for the sake of simplicity.

network. To the best of our knowledge, there was no existing example like this before. This example network serves as basis for the following Java examples as we substituted the C++ modules with Java modules.

5.2 Java INET UDP Example Networks

We provide two example Java ALSMs that can serve as basis for custom Java ALSMs.

- *jUDPEchoApp*: echoes back received UDP datagrams (similar to *UDPEchoApp* in INET).
- *jUDPBasicApp*: sends out UDP datagrams towards a remote host (similar to *UDPBasicApp* in INET)

Moreover, we provide two example networks to highlight the interoperability with existing INET UDP applications (e.g., to *UDPEchoApp*, *UDPBasicApp*). The first example network consists of two connected INET *StandardHosts*. One is running the *jUDPEchoApp* and one is running *UDPBasicApp* from INET in order to show that it is possible to access all fields from the Java code correctly. The second example network consists of two connected INET *StandardHosts*. One is running the *jUDPBasicApp* and one is running the *UDPEchoApp* from INET to show that it is possible set up packets that are understood by the *udp* module of INET. Finally, we have one network that consists of both the *jUDPBasicApp* and *jUDPEchoApp*.

5.3 Performance Evaluation

In this section, we evaluate the performance of Java ALSMs and compare them to C++ ALSMs. This is feasible as *UDPBasicApp* and *UDPEchoApp* from INET are mostly functionally equivalent to their Java counterparts *jUDPEchoApp* and *jUDPBasicApp*. We evaluate the execution time on the basis of the number of packets echoed between the ALSMs. Fig. 5 depicts the evaluation results (note that the graphs are logarithmic in the X and Y axis).

As depicted, the execution time of the simulation grows linearly with the number of echoes packets, which is the expected behavior. For 10 packets and OMNeT++ 5.0, C++ ALSMs (0.0008 s) are approx. one order of magnitude faster than Java ALSMs (0.0097 s). However, C++ ALSMs (6.89 s) are approx. twice as fast as Java ALSMs (14.5 s) for 100,000 packets. We assume that the initialization of the JVM leads to the difference for short simulation runs. In the debug mode (not depicted) and with 100,000 packets sent, C++ ALSMs (approx. 29 s) were only approx. 40 % faster than Java ALSMs (approx. 41 s) and in general much slower than both Java and C++ ALSMs in release mode. For 10 packets and OMNeT++ 5.4, C++

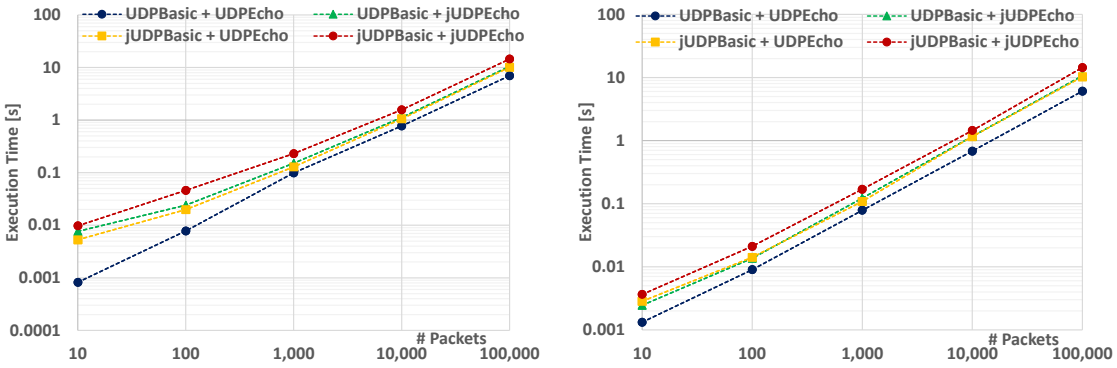


Figure 5: Execution times for different simulation setups (the graphs are logarithmic in the X and Y axis). a) OMNeT++ 5.0 b) OMNeT++ 5.4

ALSMs (0.0013 s) are approx. three times faster than Java ALSMs (0.0036 s). Again, C++ ALSMs (6.04 s) are approx. twice as fast as Java ALSMs (14.39 s) for 100,000 packets. It is remarkable that the performance of mixed language setups (one C++ ALSM and one Java ALSM) reach an intermediate performance.

Concerning the different version of OMNeT++, OMNeT++ 5.0 is faster than OMNeT++ 5.4 for short simulation runs, where 10 or 100 packets are echoed. However, OMNeT++ 5.4 is slightly faster than OMNeT++ in long simulation runs, where 10,000 or 100,000 packets are echoed.

We simulated every parameter setup 100 times and calculated the mean execution time. As simulation setup, we used two distinct configurations: OMNeT++ 5.0 with INET 3.4.0 and OMNeT++ 5.4 with INET 3.6.4. The simulations were conducted using the Cmdenv and the release mode. As hardware setup we used an i7-3770 with 32 GB RAM and SSD for the host machine. Four CPU cores and 8 GB RAM where dedicated to the guest VM.

6 Conclusion

In this paper, we contribute to the vision of a combined evaluation using both network simulations and practical testbeds. Our idea is to firstly simulate an approach using a Javal ALSM and secondly to easily derive a Java prototype implementation. This is of specific interest as the Java programming language is very suitable for rapid prototyping in practical testbeds.

In order to ease this derivation, we propose the jUDPWrapper, which enables to access the OMNeT++/INET UDP functionality from Java using a socket-based API and the same methods as on a real device (e.g., *bind()*, *send()*). Previously, the JEO supported only the access to message fields that have standard data types (e.g., int, double). In order to access the INET UDP functionality, we propose a generic approach to access message fields that have a custom type. This is important for the JEO in general as the interface towards INET is crucial for reusing existing modules and thus reducing the implementation effort. Moreover, we developed different example networks to evaluate the interoperability between Java ALSMs and INET or C++ ALSMs and provide examples for Java ALSMs (similar to *UDPBasicApp* and *UDPEchoApp* in INET) to ease the development of custom Java ALSMs. Finally, the

entire System (OMNeT++, JEO, jUDPWrapper, and examples) is publicly available. These examples are easy to use as there is no configuration effort needed. Concerning the performance evaluation, we demonstrate that C++ ALSMs can be approx. twice as fast as Java ALSMs for long simulation runs in the release mode. However, this speedup is reduced in debug mode or if a mixed language setup is used (e.g., some C++ ALSMs and some Java ALSMs).

Concerning future work, an extensions of the wrapper functionality towards TCP would be appreciated. Especially, since RFC8323 suggests to add TCP support to CoAP [3].

References

[1] HOWTO Use Linux Containers to set up virtual networks: https://www.nsnam.org/wiki/HOWTO_Use_Linux_Containers_to_set_up_virtual_networks.

[2] INET Framework for OMNeT++/OMNEST - API Reference: https://omnetpp.org/doc/inet/api-current/neddoc/index.html.

[3] C. Bormann, S. Lemay, H. Tschofenig, K. Hartke, B. Silverajan, and B. Raymor. *CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets*. RFC Editor, 2018.

[4] H. Puttnies, D. Timmermann, and P. Danielis. An approach for precise, scalable, and platform independent clock synchronization. In *2017 14th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pages 461–466, 2017.

[5] Thomas R. Henderson, Mathieu Lacage, George F. Riley, Craig Dowell, and Joseph Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 14(14):527, 2008.

[6] Bjorn Konieczek, Michael Rethfeldt, Frank Golatowski, and Dirk Timmermann. Real-Time Communication for the Internet of Things Using jCoAP. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 134–141.

[7] Jereme Lamps, David M. Nicol, and Matthew Caesar. Timekeeper: A lightweight virtual time system for linux. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 179–186, 2014.

[8] M. Rethfeldt, B. Beichler, H. Raddatz, F. Uster, P. Danielis, C. Haubelt, and D. Timmermann. Mini-Mesh: Practical assessment of a miniaturized IEEE 802.11n/s mesh testbed. In *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6, 2018.

[9] Henning Puttnies, Peter Danielis, Christian Koch, and Dirk Timmermann. Java Extensions for OMNeT++. *arXiv preprint arXiv:1709.02823*, 2017.

[10] Lorenzo Saino, Cosmin Cocora, and George Pavlou. A toolchain for simplifying network simulation setup. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, pages 82–91, 2013.

[11] Klaus Wehrle, Mesut Günes, and James Gross. *Modeling and tools for network simulation*. Springer Science & Business Media, 2010.