Cross-Platform Protocol Development Based on OMNeT++

Stefan Unterschütz
Hamburg University of Technology, Germany
stefan.unterschuetz@tu-harburg.de

Andreas Weigel
Hamburg University of Technology, Germany
andreas.weigel@tu-harburg.de

Volker Turau
Hamburg University of Technology, Germany
turau@tu-harburg.de

ABSTRACT

Software development for wireless sensor networks can be accomplished with dedicated operating systems such as Contiki or TinyOS. However, protocol design and verification as well as debugging is still challenging. On the other hand, high-level simulation environments, e.g., OMNeT++, allow convenient and rapid development, but the resulting code has to be re-implemented for a particular hardware platform. This paper introduces CometOS, a component-based, extensible, tiny operating system for wireless networks. CometOS is written in C++ and highly inspired by OMNeT++'s communication paradigm. It allows a cross-platform execution of protocols on OMNeT++ as well as on resource-restricted platforms such as wireless sensor nodes. A feasibility study is carried out on 93 nodes in the solar tower plant Jülich, Germany.

1. INTRODUCTION

Protocol design for wireless networks is a challenging task. A hardware centric development procedure is often inappropriate because of reduced debugging and testing capabilities. Additionally, the mere effort and costs to set up testbeds of meaningful size are significant—if this option is even available in the first place. Hence, simulation is an indispensable method to verify and evaluate (wireless) network protocols. High-level simulation frameworks like OMNeT++ provide powerful tools for live-inspection, visualization, configuration, etc. They are, however, usually not well suited for simulation at code-level, i.e., simulating code for a particular hardware platform. Therefore, additional effort is required to port the protocol implementation.

Obviously, sharing code among simulation and testbed environment can significantly increase the development speed. Simulators for the widely-spread TinyOS (TOSSIM, [4]) and Contiki operating systems (COOJA/MSPSim, [6]) provide this possibility. We approach the issue from the other direction and propose CometOS, a lightweight framework for the OMNeT++ simulator, enabling effortless transition to a hardware platform while maintaining most of the rich set of features of the OMNeT++ simulation environment.

CometOS is published under GPLv3 license[1]. Note that only the base framework including the MAC abstraction layer (see Sect. 3) is published. Network protocols developed in research projects are excluded. Currently supported are simulations within Atmel's ATmega128RFA1 and OMNeT++ module.

This paper is structured as follows: Section 2 briefly describes other simulation tools offering similar capabilities. Subsequently, the principles and architecture of CometOS are introduced and the programming interface is illustrated by examples. Section 4 discusses our approach and demonstrates its feasibility in simulations and a testbed. The last section concludes our approach and points at future work.

OMNeT++ 2012, March 19-23, Desenzano del Garda, Italy

DOI 10.4108/icst.simutools.2012.247711

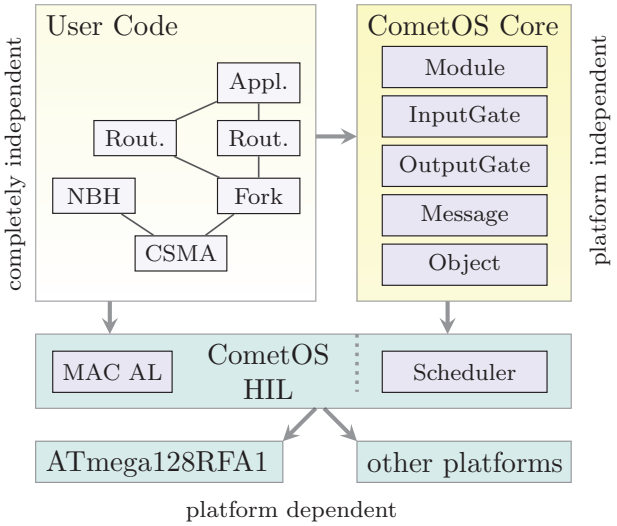
2. RELATED WORK

OMNeT++ is a object-oriented, discrete event simulation framework [7]. Its core is a module-based architecture and a message passing concept. Modules are interconnected by gates which are used as access points for messages. This approach provides a very loose coupling and greatly promotes the development of reusable building blocks. While OMNeT++ itself does not provide specific components to simulate a certain problem domain, a large number of model frameworks have emerged addressing this issue. For example, the MiXiM framework [2] aims at the simulation of wireless sensor networks and provides extensive support for modelling the wireless communication channel, mobility of nodes, MAC layers and energy consumption.

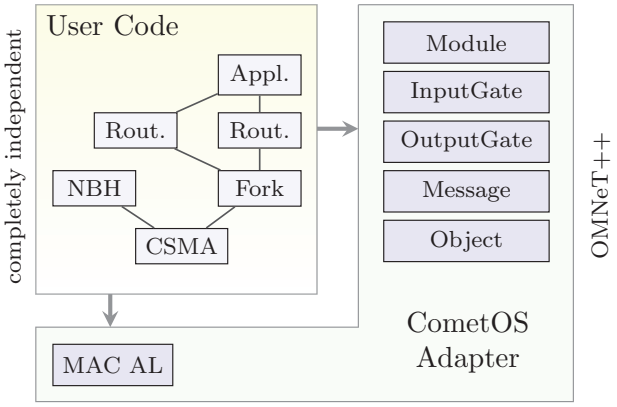
TOSSIM is a code level simulator for TinyOS applications written in the nesC programming language [1, 5]. It provides physical layer modelling at Bit granularity, inspection of base-type variables and the visualization tool TinyViz. Experiments are configured using the Python scripting language. Some drawbacks of TOSSIM are the limitation to a single code image and limited debugging capabilities—as there is no debugger for nesC, one has to debug the generated C-Code and mentally translate identifiers. This can be hard especially for instances of generic components, which are distinguished only by unique arbitrary number prefixes.

Another popular OS for wireless sensor networks, Contiki, comes with the COOJA/MSPSim simulator [6]. Its highlight is the so-called cross-level simulation. Thereby, nodes within the same network can be simulated at different levels: at network level (a high-level Java implementation), at operating system level (an actual Contiki-application) and at

[1] www.ti5.tu-harburg.de/research/cometos/cometos.zip



(a) Hardware Platform



(b) OMNeT++ environment

Figure 1: CometOS Architecture

machine code instruction level (a per-instruction hardware emulation). Additionally, COOJA also enables the simulation of TinyOS code.

3. ARCHITECTURE

A key design goal for CometOS is to provide a lightweight, flexible and extensible framework for a rapid network protocol development. Most concepts are directly adopted from OMNeT++. The core of CometOS comprises the classes `Module`, `InputGate`, `OuputGate`, `Message` and `Object`. These implement the basic concepts of OMNeT++: message passing via input and output gates, scheduling of self-messages and attaching arbitrary objects to message instances.

As shown in Fig. 1, the realization of CometOS is split into two major parts. The upper and lower figures show the architecture for a target hardware and OMNeT++, respectively. Both have in common that user protocols can be implemented and executed without porting effort.

Besides user code, the architecture for the actual hardware platform consists of a platform-independent core framework that implements the module and message passing concept. This requires a scheduler, which is itself part of a hardware-independent layer (HIL). Note that the scheduler interface is not directly exposed to the user code. Its functionality is only accessible through the `Module` class. Furthermore, the HIL provides high-level interfaces for accessing parts of the target hardware, e.g., the medium access control abstraction layer (`MAC AL`). Details on the implementation of CometOS for a target platform are given in Sect. 3.5.

For the simulation of user code the functionality of CometOS is mapped to the infrastructure of OMNeT++ using the class adapter pattern. That way, simulations of protocols written for CometOS do not differ from dedicated OMNeT++ simulations. Runtime support for inspection of modules and messages, location of undisposed objects and eventlogging is maintained. If needed, models for simulating hardware-specific components have to be implemented (e.g., `MAC AL`).

```
1  using namespace cometos;
2
3  class MyMessage: public Message {
4  };
5
6  class MySender: public Module {
7  public:
8    OutputGate<MyMessage> gateOut;
9
10   MySender() :
11     gateOut(this, "gateOut")
12   {}
13
14   void initialize() {
15     schedule(new Message, &MySender::traffic);
16   }
17
18   void traffic(Message* timer) {
19     gateOut.send(new MyMessage);
20     schedule(timer, &MySender::traffic, 500);
21   }
22 };
23
24 class MyReceiver: public Module {
25 public:
26
27   InputGate<MyMessage> gateIn;
28
29   MyReceiver() :
30     gateIn(this, &MyReceiver::handle, "gateIn")
31   {}
32
33   void handle(MyMessage *msg) {
34     // process message reception
35     delete msg;
36   }
37 };
```

Listing 1: Gates and Message Passing

In comparison to OMNeT++, CometOS provides less rich functionality due to the need of running it on resource-constrained microcontrollers, e.g., only a clock accuracy of milliseconds is currently provided by each `Module`. In the following the main principles are introduced in more detail.

3.1 Message Passing Interface

CometOS adopts the concept of gates for inter-module communication from OMNeT++. Similarly, we use a synchronous execution model: events like message reception are modelled as non-preemptive tasks. The order of execution is governed by the task scheduler at the target platform and by the OMNeT++ event scheduler within the simulation environment.

As an extension to the message passing system, we added type safety and handler methods. Gates are instantiated for a subclass of `Message`. A connection between an input and an output gate is only possible if they define the same type. This reflects the idea that a gate defines a specific interface. Furthermore, dynamic casting on the microcontroller is avoided, which would otherwise need C++'s RTTI. We also believe that explicit type safety for messages makes the development of protocols less error-prone.

OMNeT++ forwards all messages to the method `handleMessage()`. In complex modules, this method often degenerates to a simple dispatcher. CometOS binds handlers directly to gates. The handler itself must provide a signature corresponding to the message type of the gate. That way, explicit dispatching and type casting is avoided. Self-messages can be directly scheduled in combination with a handler, thus multiple timers mapped to different handlers can easily be implemented.

Listing 1 shows how to define input and output gates within modules and how to register the corresponding handler methods. In line 3, a subclass of `Message` is defined. Here, no members are defined. Those can be added manually as needed—OMNeT++'s domain specific language for message definitions is not supported. Then two modules are defined. The initialization of the defined gates has to be done in the constructor. There, a pointer of the owner, a name, and, in case of an input gate, a handler must be given. The `Sender` starts a timer in the method `initialize`. When the timer fires, an instance of `MyMessage` is sent and the timer is rescheduled. At the other end, the Receiver accepts messages of type `MyMessage` and passes them to the method `handler()`.

CometOS already proposes a convention for communication between neighboring protocols. Modelled after the primitives defined for service access points, we add the basic message types `DataRequest` and `DataIndication` and some base modules for communication endpoints and layers. These provide a convenient basis for the implementation of protocol stacks.

3.2 MAC abstraction layer & Airframes

In order to enable wireless communication, platform-dependent code for accessing the wireless channel is necessary. Within the OMNeT++ simulation environment this can be provided, e.g., by the MiXiM framework [2], whereas hardware nodes employ driver software for the used transceiver chip. An important design goal of CometOS is to keep the portion of platform-dependent code as small as possible. For this purpose we introduced a MAC abstraction layer which serves as basis for higher-level protocols. This generic layer has a rich interface for sending and receiving data and already provides a configurable CSMA-CA medium access protocol with support for random backoffs, clear channel assessment and acknowledgements and retransmissions. The rationale for choosing this level of abstraction is to match the hardware-software-boundary of the abundant 802.15.4-compliant devices, which offer integrated support for those mechanisms. By providing the possibility to disable any of those functions individually, the MAC abstraction layer may on the other hand also serve as a basis for other medium access schemes like TDMA and low-power-probing or -listening.

```
1 uint16_t val1=10, val2=20;
2 request->getAirframe() << val1 << val2;
3 ...
4 uint16_t val1, val2;
5 indication->getAirframe() >> val2 >> val1;
```

Listing 2: Airframes

Data packets for the wireless communication are represented by the class `Airframe` since OMNeT++'s `cPacket` is not suited for the realization on a microcontroller. `Airframe` is a managed byte array supporting serialization and deserialization of data. Airframes are part of the message types `DataRequest` and `DataIndication`. The code snippet in Listing 2 demonstrates the usage of airframes.

Airframes support serialization and deserialization for all native types. For user-defined types, e.g., structs, this serialization has to be implemented by the user. This provides the opportunity to pack the data very tight. A serialization using the `memcpy` function is not recommended since endianness, padding, and packing of types may differ if heterogeneous platforms are used within a network.

3.3 Cross-Layer Support

In general, CometOS is intended for a stacked protocol design. However, in some situations cross-layer approaches might be desired. For this purpose we added the possibility of loose object aggregation. Each message can be associated with additional objects deriving from the class `Object`. Note that a similar kind of object aggregation is utilized by NS3 [8]. The ownership handling of aggregated objects is similar to OMNeT++'s `ControlInfo` objects.

```
// Application: set tx power to -20 dBm
request->add(new MacTxPower(-20));
...
// MAC: use MacTxPower if set
MacTxPower* txPower= request->get<MacTxPower>();
if (txPower != NULL) {...}
```

Listing 3: Cross Layer Support

Listing 3 shows how to use object aggregation in CometOS. In the example an application protocol sets the transmission power for a data request. The connected MAC layer checks whether a message is associated with additional control information.

3.4 Configuration and Initialization

The configuration of a node, meaning the instantiation of modules, setting parameters and connecting gates, differs between the OMNeT++ and a target platform. Listing 4 shows configuration code for the sender and receiver modules in Sect. 3.1.

```
1 // Setup for OMNeT++ in NED language
2 // (skipped declaration of modules)
3 network Network {
4     submodules:
5         sender: MySender;
6         receiver: MyReceiver;
7     connections:
8     sender.gateOut --> receiver.gateIn;
9 }
10
11 // Setup for AVR
12 MySender sender;
13 MyReceiver receiver;
14
15 int main() {
16   sender.gateOut.connectTo(receiver.gateIn);
17   cometos::initialize();
18   cometos::run();
19   return 0;
20 }
```

Listing 4: Configuration for OMNeT++ and AVR

While the former is done using the NED language and `.ini` files as ever (see lines 1-9), the latter requires a C++ configuration file containing the function `main`. This procedure is straightforward as depicted in the lines 12-20.

3.5 Additional Target Platforms

Currently, CometOS is implemented for the OMNeT++ simulation environment and the ATmega128RFA1 microcontroller with an integrated 802.15.4-compliant transceiver. Support for an platform based on an ARM Cortex M3 and a TI CC1120 transceiver is planned for near future.

To ease transition to other hardware platforms, we strove for keeping the interface to the hardware-dependent part as slim as possible. The core framework at minimum needs the implementation of a scheduler to realize message passing. As the CometOS core makes use of dynamic memory allocation, this functionality must be available as well. Furthermore, the MAC abstraction layer (see Sect. 3.2) has to be implemented to provide high-level access to the transceiver and enable transmissions to the wireless channel. Interfaces to peripherals, e.g., sensors, EEPROM, are currently not prescribed by CometOS and have to be added as needed. Note that CometOS doesn't rely on C++'s RTTI, exception handling, and the STL.

4. EVALUATION

In this section we evaluate CometOS in matters of resource demand, usability, and accuracy of the simulation. For this purpose CometOS is implemented in OMNeT++ and for an AVR 8-bit microcontroller (using GNU's C++ compiler version 4.5). For the latter we use a microcontroller (ATmega128RFA1) with an embedded 802.15.4 compliant transceiver and 16 kB SRAM, 16 MHz clock frequency, and 128 kB Flash memory.

Most findings of this section originate from the experiences gained in the HelioMesh project [3]. In this project, we evaluate the feasibility of wireless mesh networks as control technology for a field of self-powered heliostats. A testbed containing 93 wireless nodes was deployed in the heliostat field Jülich, Germany (see Fig. 2).

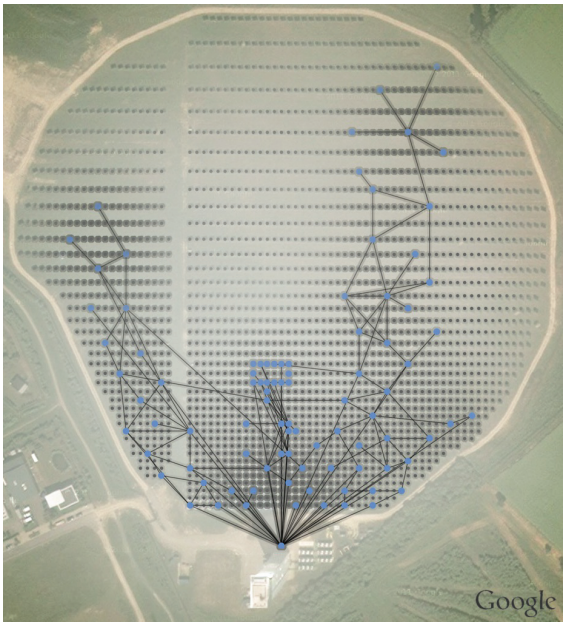


Figure 2: HelioMesh Testbed Jülich, Germany (Latitude, Longitude: 50.913316, 6.387691)

4.1 Feasibility

CometOS benefits from its close relationship with OMNeT++, which allows a convenient protocol development using the tools supported by simulator and Eclipse/CDT. Most software bugs in the HelioMesh communication stack, e.g., memory leaks, could be fixed with help of the simulator. When we deployed the code to the target platform, we only observed errors at driver level. Once the drivers were working as expected, no failures occurred that could not be detected and fixed within the simulation environment.

The HelioMesh firmware instantiates 20 modules for different purposes. Those include routing, neighborhood, clustering, and application protocols and build up a branched protocol stack. CometOS's airframe concept and the object aggregation have proven to be useful for creating complex protocol stacks. Both allow a loose coupling of modules, which highly increases the reusability of the developed protocols.

Since the resource demand in OMNeT++ is noncritical, we restrict the following analysis to the AVR implementation only. The core system and the tiny sample application (see Listings 1 and 4) needs 8.4 kB of flash memory and 478 Bytes statically allocated RAM. The firmware of the HelioMesh project (20 modules) occupies 62 kB and uses 4.5 kB static and allocates in average 1.5 kB dynamic memory. Note that the memory consumption also depends on the number of instantiated modules and the used protocols and may significantly differ for various software. However, the used controller (ATmega128RFA1) easily meets the resource demand.

4.2 Simulation Accuracy

We compared the accuracy of the simulation with the testbed in Jülich. The latter is controlled via a basestation software, written in Java, which is connected via USB to a gateway node. In OMNeT++, the complete HelioMesh

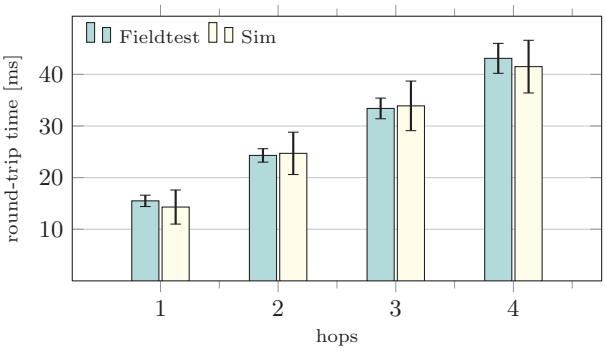


Figure 3: Comparison of Simulation and Testbed

firmware was simulated with a real-time scheduler, too. We reused the same basestation software for the OMNeT++ simulation. The network was connected to a gateway node via TCP/IP, simulating the serial connection. The wireless channel and transmission is simulated with the MiXiM framework. Its configuration is set corresponding to the used 802.15.4 transceiver hardware. For the simulated environment no overload of the real-time scheduler occured, i.e., simulation time and real time were synchronous.

We measured the round-trip time (RTT) for the communication between the basestation and nodes of various hop count. 50 Bytes payload were used for each packet. Link-layer reliability is applied, meaning that each one-hop transmission is acknowledged. The averages and standard deviations (100 measurements each) are depicted in Figure 3. In general, the results of simulation and testbed are close together. However, the higher standard deviation in the simulation are due to a timing jitter caused by the TCP/IP connector. The USB connection in the testbed shows less fluctuations. For the used protocol stack, the additional processing time caused by the execution of CometOS's scheduling and message passing mechanisms and the protocols themselves (which is not accounted for within the simulation) did not significantly bias the experiment results. The experiment shows that simulations of protocols implemented for CometOS can provide a good estimation of their real-world performance.

5. CONCLUSIONS AND FUTURE WORK

This paper introduces CometOS, a component-based, extensible, tiny operating system for wireless networks. CometOS adopts concepts of OMNeT++ and allows platform-independent protocol development. Higher level protocol implementations can be executed within the simulation environment itself as well as on resource-restricted hardware. An evaluation was done based on ATmega128RFA1 modules in a deployment of 93 nodes in the heliostat field Jülich. The experiment revealed that CometOS is competitive with COOJA and TOSSIM with regard to the simulation environment, accuracy (due to the MiXiM framework), extensibility, debugging capabilities and its functional facilities inherently given by C++.

The released software provides no communication protocols and is still work in progress. Furthermore, a generic configuration of modules is missing yet, but will be integrated using the NED language in the future. Currently, we also lack the capability to conveniently manage and configure distinct sets of statistics collection for simulation and hardware environment. CometOS is used for further research projects, e.g., a large-scale readout and configuration of electricity meters.

6. REFERENCES

[1] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A holistic Approach to Networked Embedded Systems. *SIGPLAN Not.*, 38, May 2003.

[2] A. Köpke, M. Swigulski, K. Wessel, D. Willkomm, P. T. K. Haneveld, T. E. V. Parker, O. W. Visser, H. S. Lichte, and S. Valentin. Simulating Wireless and Mobile Networks in OMNeT++ the MiXiM Vision. In *Simutools '08: Proc. first International Conference on Simulation Tools and Techniques for Communications, Networks and Systems*, Mar. 2008.

[3] S. Kubisch, M. Randt, R. Buck, A. Pfahl, and S. Unterschütz. Wireless Heliostat and Control System for Large Self-Powered Heliostat Fields. In *SolarPACES '11: Proc. 17th International Symposium on Concentrated Solar Power and Chemical Energy Technologies*, Sept. 2011.

[4] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *SenSys '03: Proc. first International Conference on Embedded networked sensor systems*, Nov. 2003.

[5] P. Levis, S. Madden, J. Polastre, R. Szewczyk, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An Operating System for Sensor Networks. In *EUSAI '04: Proc. second European Symposium on Ambient Intelligence*, Nov. 2004.

[6] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-Level Sensor Network Simulation with COOJA. In *LCN '06: Proc. 31st IEEE Conference on Local Computer Networks*, Nov. 2006.

[7] A. Varga. The OMNeT++ Discrete Event Simulation System. In *ESM '2001: Proc. 15th European Simulation Multiconference*, Prague, Czech Republic, June 2001.

[8] K. Wehrle, M. Günes, and J. Gross. *Modeling and Tools for Network Simulation*. Springer Berlin Heidelberg, June 2010.