TKN Telecommunication Networks Group

Technical University Berlin

Telecommunication Networks Group

Proceedings of the 2nd International OMNeT++ Workshop

Holger Karl

karl@ee.tu-berlin.de

Berlin, Januar 2002

TKN Technical Report TKN-02-01

TKN Technical Reports Series

Editor: Prof. Dr.-Ing. Adam Wolisz

2nd International OMNeT++ Workshop

January 8-9, 2002
Technical University Berlin
Berlin, Germany

Technische Universität Berlin

TKN Telecommunication Networks Group

UNIVERSITÄT KARLSRUHE (TH)

Universität Karlsruhe Institut für Nachrichtentechnik INT

Preface

The second International OMNeT++ Workshop took place on January 8 & 9, 2002 at the Technische Universität Berlin. Its target was to bring together active users and developers of the OMNeT++ simulation tool in true tradition of a workshop: to provide a forum for discussions, interactions, exchange of ideas, users' wishes and hopes for the future development of OMNeT++. At this time, OMNeT++ enjoys growing popularity among a wide range of scientific communities and it competes well in these communities with established tools. It is the hope of this workshop and its organizers to foster this success story even further.

Contained within these proceedings you will find the papers and (partially) slides that were presented at the workshop. Unfortunately, you will not find the discussions that take place during the two days — plenty of time has been set aside for them, and I am sure that they will be both interesting and challenging.

I hope that you will enjoy the workshop and your stay here in Berlin!

January 2002 Holger Karl

Organization

The second International OMNeT++ Workshop was jointly organized by the Telecommunication Networks Group, Technische Universität Berlin and by the Institut für Nachrichtentechnik, Universität Karlsruhe (TH).

Executive Committee

Ulrich Kaage	Universität Karlsruhe (TH)	`kaage@int.uni-karlsruhe.de`
Holger Karl	Technische Universität Berlin	`karl@ee.tu-berlin.de`
Guenter Schaefer	Technische Universität Berlin	`schaefer@ee.tu-berlin.de`
Andras Varga	Budapest University of Technology and Economics	`andras@whale.hit.bme.hu`

An on-line version of these proceedings can be found at the website of the Telecommunication Networks Group, TU Berlin at `http://www-tkn.ee.tu-berlin.de`.

Table of Contents

Session 1:
Simulating Large and Specific Networks

A Simulation Suite for Accurate Modeling of IPv6 Protocols

Johnny Lai[1], Eric Wu[1] Andràs Varga[2], Y. Ahmet Şekercioğlu[1], and Gregory K. Egan[1]

[1] Centre for Telecommunications and Information Engineering, Monash University, Melbourne, Australia
[2] Department of Telecommunications, Technical University of Budapest, Budapest, Hungary

Abstract. As part of our ongoing research program on performance analysis of protocols for mobility management in IPv6 networks, we have developed a set of OMNeT++ models for accurate simulation of IPv6 protocols. Our simulation set models the functionality of the RFC 2373 *IP Version 6 Addressing Architecture* [5], RFC 2460 *Internet Protocol, Version 6 (IPv6) Specification* [3], RFC 2461 *Neighbor Discovery for IP Version 6 (IPv6)* [7], RFC 2462 IPv6 *Stateless Address Autoconfiguration* [10], RFC 2463 *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification* [2], and RFC 2472 *IP Version 6 over PPP* [4].

1 Introduction

Inevitably, telecommunication networks are increasingly becoming more complex as the trend towards the integration of telephony and data networks into integrated services networks gains momentum. It is expected that these integrated services networks will include wireless and mobile environments as well as wired ones. As a consequence of the rapid development and fusion of communication technologies, understanding the dynamic interaction of protocols and performance analysis are becoming much more complex to be investigated in small-scale experimental testbeds. Analytical analysis is also not feasible for similar reasons. Simulation is now considered as a tool of equal importance (as complementary to the analytical and experimental studies) for investigating and understanding the behavior of complex systems.

As part of our ongoing research programs on analysis of protocol performance on mobile IPv6 networks, we have developed a set of OMNeT++ models for accurate simulation of IPv6 protocols. We have chosen OMNeT++ as the simulation framework because of the following reasons: (a) It allows the design of modular simulation models, which can be combined and reused flexibly; (b) It is possible to compose models with any granular hierarchy; (c) OMNeT++ is open-source, free for non-profit use, and has a fairly large and active user community; (d) It has support for parallel simulation; and (e) Its performance is comparable to commercial simulation tools. Section 2 presents a brief summary of the features of OMNeT++.

Our IPv6 simulation model suite consists of several functional blocks. There is also dual-stack support for analysis of protocol interactions in mixed IPv4-IPv6 networking environments. The accuracy of the simulation is ensured because of the fine-grained level of details in the simulation. Realistically formatted protocol data units (PDUs) are passed between simulated network entities and service data units (SDUs) exchanged between the adjacent protocol layers. The IPv6 datagram format currently includes most of the extension headers except the ones related to Authentication and Encapsulating Security Payload. Real data from our network testbed is used to calibrate the model, and simulated processing delays are introduced where necessary to account for the differences without sacrificing performance. The structural breakdown of the model and module descriptions can be found in Section 3.

Currently we are working on mobility support, and future enhancements in the pipeline include profiling the model for very large scale network simulations (i.e. 10,000+ network entities) and dynamic creation of network topologies through XML (extensible markup language) based configuration files.

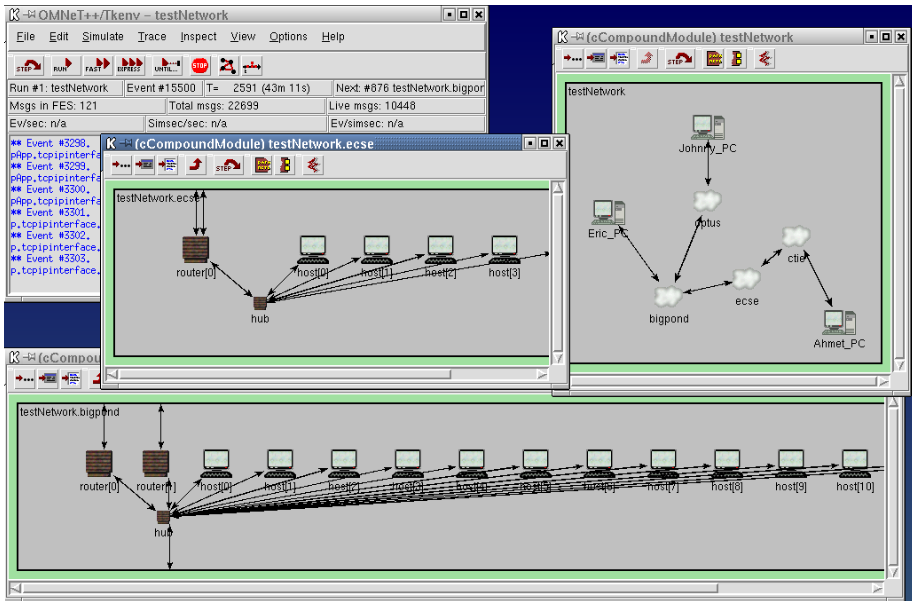
2 OMNeT++ Simulation Framework

OMNeT++ is a C++-based discrete event simulation package developed at the Technical University of Budapest by András Varga [8,12]. The primary application area of OMNeT++ is the simulation of computer networks and other distributed systems. It is open-source, free for non-profit use, and has a fairly large and active user community. It also allows the design of modular simulation models, which can be combined and reused flexibly. Additionally, OMNeT++ allows the composition of models with any granular hierarchy. It has been shown that this simulation framework is suitable for simulation of complex systems like Internet nodes and dynamics of TCP/IP protocols realistically [6,14].

Simulated models are composed of hierarchically nested modules. In OMNeT++, there are two types of modules: simple and compound modules. Simple modules form the lowest hierarchy level and implement the activity of a module, and they can arbitrarily be combined to form compound modules. Modules communicate with message passing. Messages can be sent either through connections that span between modules, or directly to their destination modules. The user defines the structure of the model (the modules and their interconnection) by using the topology description language (NED) of OMNeT++ [11].

Simple modules are implemented in C++, using the simulation kernel system calls and the simulation class library. For each simple module, it is possible to choose between process-style and protocol-style (state machine) modeling. Therefore, different parts of computing and communication systems can be programmed in their natural way and connected easily. The simulation class library provides a well-defined application programmer's interface (API) to the most common simulation tasks, including: random number generation; queues, arrays and other containers; messages; topology exploration and routing; module creation and destruction; dynamic topologies; statistics; density estimation (including histograms, P2 and k-split [13]); output data recording. The object-oriented approach allows the flexible extension of the base classes provided in the simulation kernel.

Model components are compiled and linked with the simulation library, and one of the user interface libraries to form an executable program. One user interface library is optimized for command-line and batch-oriented execution, while the other employs a graphical user interface (GUI) that can be used to trace and debug the simulation (as an example, Figure 1 shows a simulated network configuration). The GUI makes the internals of a simulation model fully visible: it displays the network graphics, animates the message flow and lets the user peek into objects (messages, queues, etc.) within the model. It is also possible to change parameters and message fields for debugging purposes. Visualization features make OMNeT++ suitable also for educational or demonstration purposes. Because of the modular design, it is possible to embed the simulation engine (including models) into other applications. OMNeT++ also has support for parallel discrete event simulations (PDES).



```
Run #1: testNetwork   Event #15500   T=   2591 (43m 11s)   Next: #876 testNetwork.bigpor
Msgs in FES: 121   Total msgs: 22699   Live msgs: 10448
Ev/sec: n/a   Simsec/sec: n/a   Ev/simsec: n/a

** Event #3298.
pApp.tcpipinterfa
** Event #3299.
pApp.tcpipinterfa
** Event #3300.
pApp.tcpipinterfa
** Event #3301.
p.tcpipinterface.
** Event #3302.
p.tcpipinterface.
** Event #3303.
p.tcpipinterface.
```

Fig. 1. OMNeT++ screen showing a hierarchical network model. The upper right window shows the topology of a simulated network consisting of four subnets. Topologies of the two of these subnets, `ecse` and `bigpond` are shown in the middle and bottom windows respectively.

3 IPv6 Simulation Model

The IPv6 simulation model suite consists of several functional blocks. As one can expect, the major blocks reside in the network and data link control layers. These blocks can be connected together to form simulated hosts, routers, Ethernet hubs, point-to-point links etc. Figure 2 shows these blocks in a model of a router with three network interfaces. The core module (`IPProcessing`) of the network layer (Figure 2(b)) provides dual-stack support (IPv4 and IPv6).

Our simulation model provides enhancements to the existing OMNeT++ IPv4 models mainly in the areas of providing interchangeable network interfaces for simulating IP protocols using various physical transport mechanisms (point-to-point links, Ethernet connections etc.). The enhancements also include the ability to model nodes having any combination of these physical devices.

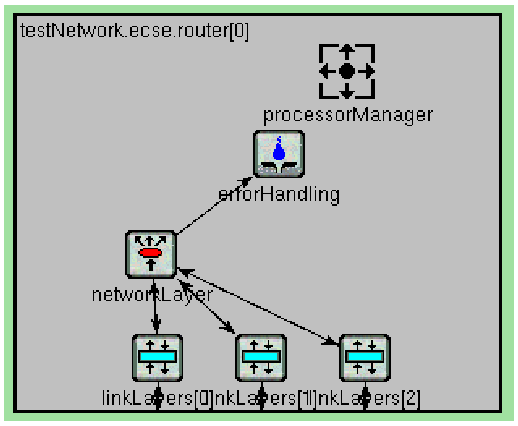
A router in an IPv6 network has many configurable parameters. We believe that the user should not have to learn the custom syntax of a configuration file in order to change a single parameter. From a user's point of view any approach that can reduce the learning curve involving a new simulation tool will be very useful. For this reason, we have chosen *Extensible Markup Language* (XML) as the format for the configuration file of the network nodes. The reasons behind our decision can be summarized as follows: XML is easy to comprehend, non-proprietary and mature technology with many tools available (parsers, viewers and validators etc.).

3.1 IPv6 Node Hierarchy

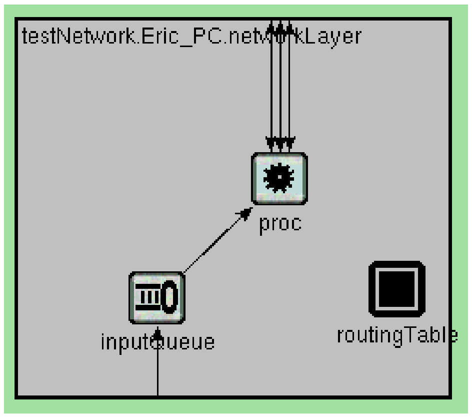
The architectural framework of the IPv6 simulation model is based on the structure of the OMNeT++ IPv4 Protocol Suite [14]. The IPv4 suite consists of modules that model the data link control, network and transport (TCP and UDP) layers. Our IPv6 simulation model framework is interoperable with the IPv4 models to support modeling dual stack routers which allow IPv4 and IPv6 packet flows simultaneously. The suite also allows various data link control layer network interfaces to be present within a single node. Therefore, it is possible to investigate the interactions between IPv4 and IPv6 protocols in a mixed protocol environment. We believe that the introduction and integration of IPv6 into the current global IPv4 infrastructure will raise performance issues that need to be investigated in large scale simulated networking scenarios.

Network Layer We have adopted a different approach than the design of the IPv4 Protocol Suite [14], and separated the network interface from the network layer. This approach has allowed us to add new models of physical interfaces and to simulate routers that can have a combination of various network cards. In our simulation suite, the network layer contains only the IP processing, IP input queue and the IPv4 routing table modules (Figure 2(b)). The main functional blocks of the IP processing module are as follows (see Figure 3): The IP discriminator (`ipd`) module checks the IP version and forwards the packets to the correct IP stack. The IP combine (`ipc`) receives the packet from either the IPv4 or IPv6 stack and forwards the packet to the data link control layer.

Data Link Control Layer The Data link control layer module shown in Figure 4 contains the input queue and an interchangeable network interface. This arrangement allows one to accommodate different physical transports without a need of recompilation of simulation models. At the time of writing, PPP and Ethernet interfaces have been implemented. The Ethernet model also includes a hub which is derived from the work of Baresi [1].



(a) Top level modules.



(b) Structure of the Network Layer module.

Fig. 2. The simulation model of a router with three network interfaces.

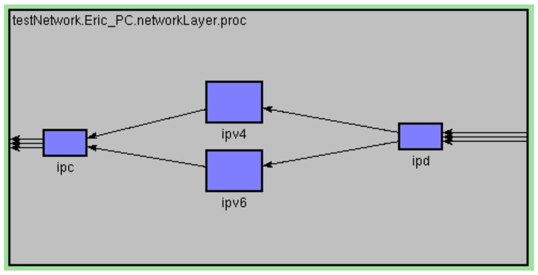


Fig. 3. The main functional blocks of the IP processing module (which is part of the network layer shown in Figure 2(b)). See text for details.

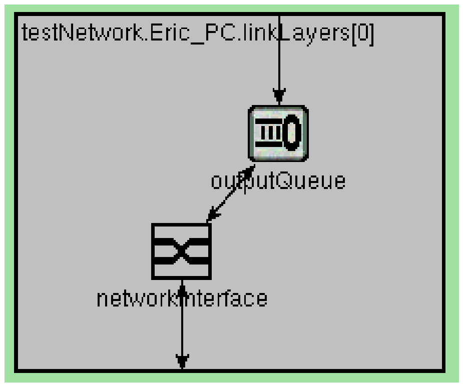


Fig. 4. Structure of the simulation model for generic data link control layer modules.

3.2 Processing IPv6 Datagrams

The core functionality of the IPv6 Simulation model is implemented in the IPv6 processing compound module (`ipv6` block in `proc` which is part of the network layer). See the Figures 2(b), 3 and 5. This module determines the destination of packets, initiates and receives ICMP notifications, and implements Neighbour Discovery mechanisms.

Referring to Figure 5, the compound module *IPv6Processing* consists of the following submodules: *PreRouting6* (`preRouting`), *IPv6LocalDeliver* (`localDeliver`), *Routing6* (`routing`), *IPv6Multicast* (`multicast`), *AddressResolution* (`addrResln`), *ICMPv6* (`ICMP`), *IPv6Send* (`send`), *IPv6Output* (`output`), *IPv6Fragmentation* (`fragmentation`) and *RoutingTable 6* (`RoutingTable6`).

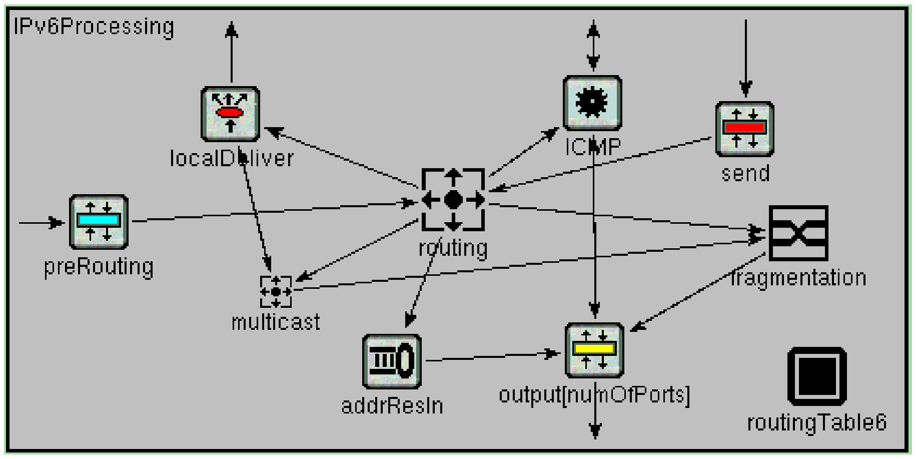


Fig. 5. Internal structure of the compound module *IPv6Processing*.

Datagrams arriving a node will encounter the *PreRouting6* module first. In this module, a hook can be implemented to gather statistics or filter packets as described in [14]. Next hop determination is the responsibility of the *Routing6* module. Its options are:

- Send the datagram to an output interface via the fragmentation module when forwarding of packets is in effect i.e., it is a router,
- Send the datagram to multicast module when the packet has a multicast destination address,
- Send the datagram to localDeliver module for local delivery of the datagram.

LocalDeliver accepts datagrams destined for the local node, decapsulates the datagram and delivers its contents to upper layers. Any destination options encountered in the datagram are also processed here.

The *AddressResolution* module queries neigbhours for their data link control layer address and responds to the same requests issued by neighbouring nodes. It aims to follow the prescribed procedures defined in RFC 2461 [7] as closely as possible.

Determination of the next hop neighbour is accomplished in `Routing6` as mentioned previously with the aid of the simple module *RoutingTable6*, which contains the conceptual data structures mentioned in Section 5.2 of RFC 2461 [7]. Many other simple modules rely on *RoutingTable6* to provide access to those structures, notably *NeighbourDiscovery*, *Multicast* and *AddressResolution*.

IPv6Send encapsulates the upper layer SDUs into IPv6 datagrams and sends them to *Routing6* for further processing.

The *IPv6Fragmentation* module accepts outgoing datagrams from *Routing6* and checks to see if fragmentation is required before transferring the packets to *IPv6Output* module.

ICMP packets are managed by the *ICMPv6* compound module. The internal structure of this module is shown in Figure 6. It contains three simple modules *ICMPv6Core*, *NeighbourDiscovery* (`nd`) and *ICMPCombine* (`combine`). The *ICMPv6Core* module implements most of the RFC 2463 [2].

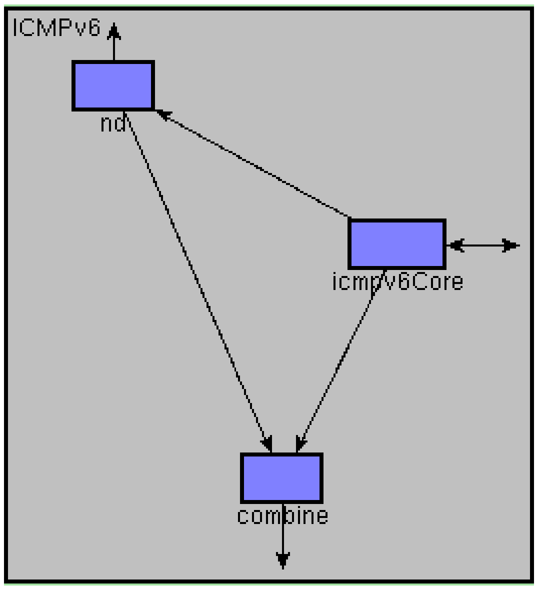


Fig. 6. Components of the ICMP compound module.

The *NeighbourDiscovery* simple module initiates and responds to neighbour discovery messages according to the role of the node (host or router) in conformance with RFC 2461 [7]. *AutoConfiguration* has also been added in accordance with RFC 2462 [10].

The accuracy of the simulation is ensured due to the fine-grained level of detail in the simulation. Datagrams are passed between network entities and SDUs exchanged between the adjacent protocol layers. The IPv6 datagram currently implements most of the extension headers mentioned in [3] except the Authentication and Encapsulating

Security Payload headers. Real data from simple network testbed is used to calibrate the model. Simulated processing delays are introduced where necessary to account for the differences without sacrificing performance.

3.3 Node Configuration and Parameter Specification Files

The network configuration (i.e. the connections between the network entities) is described through OMNeT++'s NED language. In addition to this, for each IPv6 node, a set of parameters can be configured by writing an XML document. (A sample XML document is shown in Figure 7). There are two ways to configure parameters of a node. In the `<global>` section, a parameter for all interfaces of all nodes on the same network is set, and in the `<local>` section, a particular parameter on a specific interface of the node is set.

4 Concluding Remarks and Future Work

Future enhancements in the pipeline include adding support for mobility; profiling the model for very large scale network simulations (i.e. 10,000+ network entities) and dynamic creation of network topologies through XML configuration file.

5 Acknowledgment

This work is supported through a Victorian Partnership for Advanced Computing (VPAC) expertise grant.

References

1. M. Baresi. `EtherDemo` – a simple ethernet (802.3) simulation. URL reference: http://whale.hit.bme.hu/cgi-bin/contrib.pl?dir=models&txt=EtherDemo-1.0.
2. A. Conta and S. Deering. RFC 2463 Internet Control Message Protocol (ICMPv6) for the Interent Protocol Version 6 (IPv6) Specification, 1998. URL reference: http://www.faqs.org/rfcs/rfc2463.html.
3. S. Deering and R. Hinden. RFC 2460 Internet Protocol, Version 6 (IPv6), 1998. URL reference: http://www.faqs.org/rfcs/rfc2460.html.
4. D. Hasken and E. Allen. RFC 2472 IP Version 6 over PPP, 1998. URL reference: http://www.faqs.org/rfcs/rfc2472.html.
5. R. Hinden and S. Deering. RFC 2373 IP Version 6 Addressing Architecture, 1998. URL reference: http://www.faqs.org/rfcs/rfc2373.html.
6. U. Kaage, V. Kahmann, and F. Jondral. An OMNeT++ TCP model. In *Proceedings of the European Simulation Multiconference (ESM'2001)* [9].
7. T. Narten, E. Nordmark, and W. Simpson. RFC 2461 Neigbhour Discovery for IP Version 6 (IPv6), 1998. URL reference: http://www.faqs.org/rfcs/rfc2461.html.
8. OMNeT++ object-oriented discrete event simulation system. URL reference: http://www.hit.bme.hu/phd/vargaa/omnetpp.htm, 1996.
9. The Society for Modeling and Simulation International (SCS). *Proceedings of the European Simulation Multiconference (ESM'2001)*, Prague, Czech Republic, June 2001.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE netconf SYSTEM "netconf.dtd">

<netconf>
  <global>
    <gAdvSendAdvertisements>on</gAdvSendAdvertisements>
    <gMaxRtrAdvInterval>1700</gMaxRtrAdvInterval>
    <gMinRtrAdvInterval>500</gMinRtrAdvInterval>
    <gAdvManagedFlag>on</gAdvManagedFlag>
    <gAdvOtherConfigFlag>on</gAdvOtherConfigFlag>
    <gAdvLinkMTU>5644</gAdvLinkMTU>
    <gAdvReachableTime>33567</gAdvReachableTime>
    <gAdvRetransTimer>5346</gAdvRetransTimer>
    <gAdvCurHopLimit>457457</gAdvCurHopLimit>
    <gAdvDefaultLifetime>8000</gAdvDefaultLifetime>
    <gHostLinkMTU>1400</gHostLinkMTU>
    <gHostCurHopLimit>2</gHostCurHopLimit>
    <gHostBaseReachableTime>232</gHostBaseReachableTime>
    <gHostRetransTimer>234</gHostRetransTimer>
    <gHostDupAddrDetectTransmits>1</gHostDupAddrDetectTransmits>
  </global>
  <local node="host1">
    <interface>
      <inet_addr scope="global">435:345:4:0:260:97ff:0:1/64</inet_addr>
    </interface>
  </local>
  <local node="router">
    <interface>
      <inet_addr scope="link">fe80:0:0:0:260:97ff:0:5/64</inet_addr>
      <AdvSendAdvertisements>off</AdvSendAdvertisements>
      <MaxRtrAdvInterval>1231</MaxRtrAdvInterval>
      <MinRtrAdvInterval>344</MinRtrAdvInterval>
      <AdvManagedFlag>off</AdvManagedFlag>
      <AdvOtherConfigFlag>off</AdvOtherConfigFlag>
      <AdvLinkMTU>250</AdvLinkMTU>
      <AdvReachableTime>1888</AdvReachableTime>
      <AdvRetransTimer>222</AdvRetransTimer>
      <AdvCurHopLimit>10</AdvCurHopLimit>
      <AdvDefaultLifetime>6710</AdvDefaultLifetime>
      <AdvPrefixList>
        <AdvPrefix>3018:FFFF:0:0:0:0:0:0/48</AdvPrefix>
      </AdvPrefixList>
      <HostLinkMTU>60</HostLinkMTU>
      <HostCurHopLimit>5</HostCurHopLimit>
      <HostBaseReachableTime>500</HostBaseReachableTime>
      <HostRetransTimer>400</HostRetransTimer>
    </interface>
  </local>
</netconf>
```

Fig. 7. An example of a configuration file used for specifying the values of several parameters of the nodes in an IPv6 network.

10. S. Thomson and T. Narten. RFC 2462 IPv6 Stateless Address Autoconfiguration, 1998. URL reference: http://www.faqs.org/rfcs/rfc2462.html.
11. A. Varga. *OMNeT++ User Manual*. Department of Telecommunications, Technical University of Budapest, 1997. URL reference: ftp://ftp.hit.bme.hu/sys/anonftp/omnetpp/doc/usman.pdf.
12. A. Varga. The OMNeT++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)* [9].
13. A. Varga and B. Fakhamzadeh. The K-Split algorithm for the PDF approximation of multidimensional empirical distributions without storing observations. In *Proceedings of the* 9*th* *European Simulation Symposium (ESS'97)*, pages 94–98, Passau, Germany, October 1997. The Society for Modeling and Simulation International (SCS).
14. K. Wehrle, J. Reber, and V. Kahmann. A simulation suite for internet nodes with the ability to integrate arbitrary quality of service behavior. In *Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS'2001)*, Phoenix, Arizona, USA, January 2001.

A simulational study of multiexchange circuit switching networks (with dynamic routing) using OMNeT++

Luísa Jorge
Polytechnic Institute of Bragança
and INESC Coimbra
ljorge@ipb.pt

Paulo Melo
Faculty of Economics – University of
Coimbra and INESC Coimbra
pmelo@fe.uc.pt

1. Introduction

The work presented here is mostly a summary of the Master Thesis [Jorge2001] of one of the authors, which included the building of a simulational tool using OMNeT++ to analyse multiexchange circuit switching networks using different dynamic routing methods. The work presented intended to support the grade of service analysis regarding the traffic routing method in telecommunication networks (circuit switching networks), with a pre-determined topological structure.

In this presentation, its first described the problem solved by the simulation tool, followed by (a summary) of the implementation model used, and then by a small discussion of implementation issues created by the use of OMNeT++ and the solutions used to solve them.

2. Problem description

The work presented intended to support the grade of service (GoS) analysis regarding the traffic routing method in telecommunication networks (circuit switching networks), with a pre-determined topological structure. The performance measure used was the blocking probability.

A multiexchange circuit switching network can be characterized by the nodes (switches), the arcs (trunks) that link them, the traffic that if offered switch to switch (point to point) and the routing rules used. Several parallel channels (circuits) form each trunk. The traffic flow is the call flow between a determined origin/destination pair. A network supporting several kinds of traffic flows, (also called several call classes) each with its own traffic characteristics, can model, in teletraffic engineering an integrated services network.

A call originating in switch *O* destined to switch *D* needs, to be successful, a path (or route) of single or multiple trunk between *O* and *D* switches, with a number of free circuits in each trunk of the path between those switches at least equal to the number of circuits needed to the call. Those circuits will be occupied by the call in the selected path, for the full duration of the call, and will be released at its end. Single trunk paths are those formed by the direct arc connecting origin and destination. A path that uses transit switches (intermediate, or tandem switches), therefore one using two of more trunks its called a multiple trunks path.

When a call can be offered to several (two or more) paths (named alternate routes) according to predetermined rules, we have an alternative routing method. In dynamic alternative routing methods, call are routed in the "best" available path, according to some criteria that form the essence of each method, trying to used adequately the full extra capacity present in the telecommunication network.

In the present work, a teletraffic simulation tool was developed, to allow the analysis of GoS in circuit switching networks integrating several service classes. The simulation tool developed is generic and routing method independent. Some modules to simulate the dynamic routing methods DCR [Girard90], DAR [Mitra91] and RTNR [Ash98] (and the fixed routing method FAR [Mitra91]) were also implemented.

3. The implementation model

The structure of the telecommunication network model was established using the NED language. This depicts the topology of the telecommunication network to be studied. This model is based in two kinds of processes: the *generator* process and the *switch* process. An example of the model topology can be found on Figure 1.

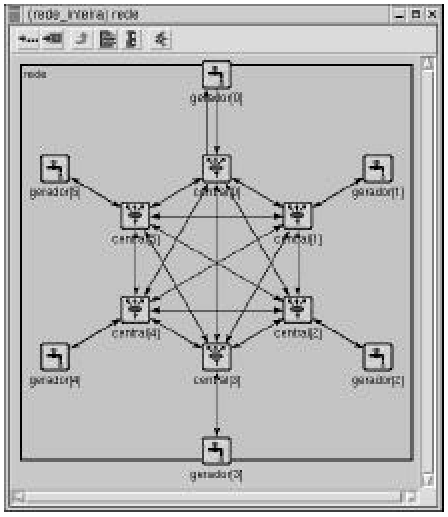


Fig. 1– Model topology (example with 6 switches).

In OMNeT++ the switching network can be modelled as a network containing several simple modules named `switch` (switching nodes) and `generator` (call generators). In the model described here, there is a `generator` for each `switch`, and the switches are fully connected with each other, with each generator connected with "its" switch (not fully connected networks can be modelled by dimensioning link capacities accordingly). The modules communicate between themselves exchanging messages. The generator modules create "calls" (which can be of several kinds), for their respective switches. The switch modules receive messages from the generators for the calls to be established. Those messages are then exchanged between switches from origin to destination, to establish the call, and back (to clean up after the call).

The several kinds of messages used can be divided in two main groups: call messages and control messages. Call messages can in turn be one of four kinds, including a message related to a call yet to be established (named "call to be established") and one corresponding to a call already ended (named "concluded call"). Control messages can be of three kinds, including an "end of simulation" message.

When the generator creates a "call to be established" message, this message is delivered to the switch connected to it. This message contains the parameters describing the call (including destination, resources needed, calling population, and others). The switch receiving it (origin switch) will now find out which path the call should follow to be established. If (as is usual) the first path to be tried is the direct route, the switch will find out whether there is free capacity on the trunk connecting the origin and destination switches. If that is possible, the message is delivered to the destination switch (and the link capacities changed), which will then process it (by changing the message kind to "concluded call" and sending it to itself, to be received when the call duration elapses). When the call ends, the destination switch sends the changed message back to the origin switch (and releases the occupation of trunks due to the call then ended).

If the routing mechanism gives the origin switch a path using a tandem switch, the origin will deliver the "call to be established" message to this intermediate switch, provided there is capacity in their common link (and changing the capacities to reflect the calls to be established). This switch will then try to deliver the call message to the destination switch, again noting the link capacity and changing it accordingly. The destination switch upon receiving the message will then proceed as in the previous example, but now will return the message not directly to the origin switch, but back to the intermediate switch (so the resources used can be released along the path). Several other (more complex) situations can occur (like when there its found in the intermediate switch that there aren't enough resources to conclude the call establishment via that route), but are dealt with by means similar to the presented, with the exchange of messages.

The generator main characteristics are its capacity to model several kinds of traffic flow, with several kinds being active in the same model, using different parameters for the stochastic processes of call arrival and ending. It can generate traffic according to a Poisson distribution (modelling infinite populations) and according to Engset distribution (modelling a finite population). The characteristics of the flow can include the traffic intensity, the size of the calls (in circuits used), and also (for Engset traffic) the number of traffic sources and the traffic intensity by free source. As said, a single generator can provide simultaneously different kinds of traffic, each with different flow characteristics.

A switch ("central") must find out the path to be used to deliver each call to be established to the destination switch. It must record each event, namely those creating changes on the network status, so they can be analysed on the end of the simulation. It is responsible to exchange the call establishing messages from the origin to the destination (possibly via tandem switches) and recording the changes on the capacity of the trunks used to establish that call. Upon completion of the call, those changes are undone by returning another message on the reverse path. The switch must also keep the information to be used on routing decisions current, according to the information available to it. Most of these tasks aren't completely routing method independent, and as such it uses auxiliary classes to perform those tasks.

The generator and the switch are independent of the routing method used. To support several different routing methods, three auxiliary classes were defined (router, "controlador" – controller, and stats). The router class function is to supply to the switch the path to be used to route a call to be established by it. The controller class is used by some routing methods to aggregate information regarding the status of the network and provide it on demand to the router class. Finally, the stats class is used to provide a single data collection and aggregation point to the changes on the network (like link capacities changes), so that several statistic measures can be computed at the end of the simulation. All these classes can be method dependent and as such can be specialized by each particular routing method used, while the switch and generator modules remain the same. A global vision of the core classes used can be found on Figure 2.

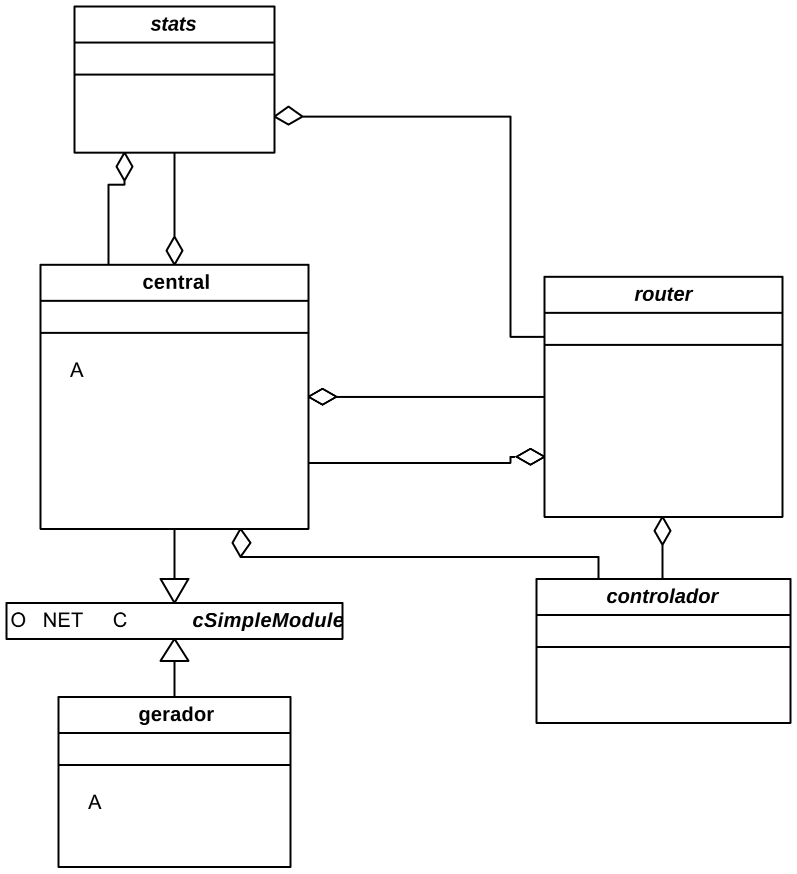


Fig. 2– Model main classes static diagram.

4. OMNeT++ related implementation issues

The model described needed a large number of random number sources to provide randomness to the different tasks (mostly on the generator module, on the call generation task, and particularly on the Engset traffic generation). As such, it was needed a la rge number of independent random number generators. In OMNeT++ the number is usually bounded at a small value, and as such the source of OMNeT++ `util.h` file was changed. The `seedtool` was also changed to improve the seed generation process for the number of random generators used (4096).

The work analysed different methods at different levels of load, and using different parameters for the routing methods used. The validation method used for the results also needed a large number of independent replications to be performed. To help automate the running of the simulations, a perl tool to generate the different parameters needed to each run (for the `omnetpp.ini` file) was built. This tool would create .ini files for a combination of parameters, each run describing a particular replication using a particular set of parameters. However, the .ini file could grow to have 500 runs, and as such, changes on the `cinifile.h` source were needed to comply with such large files (large .ini files were also needed to provide fixed seeds to the large number of random number generators).

The analysis of the results (stats class) used a change d version of the cWeightedStdDev class, as the results of the evolution of a value weren't correct with the original. The results were output in scalar output, and later filtered and input to a database, for posterior analysis.

The work was started using OMNeT++ version 1.1 and evolved up to OMNeT++ v2.0beta5. The work is now in use (research) using OMNeT++ v2.0. Only small changes were needed to the code during those evolutions. However, the change of machines (from Linux to FreeBSD and from different versions of GCC) forced a general cleanup on the code.

The code implemented made use of global variables as a means of communication between processes (to store global quantities, like the network occupation status), and as such it was impossible to use the parallel capabilities of OMNeT++.

Acknowledgment

The author would like to thank her thesis adviser, José Craveirinha, for his help and support in the work presented.

References

[Ash98] Gerald R. Ash, *Dynamic Routing in Telecommunications Networks*, McGraw Hill, New York, 1998

[Girard90] André Girard, *Routing and Dimensioning in Circuit-Switched Networks*, Addison-Wesley Publishing Company, U.S.A., 1990

[Jorge01] Luísa Jorge, *Um Estudo Simulacional de Redes Inter-centrais com Encaminhamento Dinâmico – Incluindo redes com integração de serviços*, Master's Dissertation, University of Coimbra, Faculty of Sciences and Egineering, Department of Electrotechnical Engineering, Coimbra 2001 (in Portuguese)

[Mitra91] Debasis Mitra and Judith B. Seery, "Comparative Evaluations of Randomized and Dynamic Routing Strategies for Circuit-Switched Networks", IEEE Transactions on Communications, Vol. 39, Nº 1, pp. 102-116, New York, January 1991

Using Realistic Internet Topology Data for Large Scale Network Simulations in OMNeT++

Roland Bless

Institute of Telematics, Universität Karlsruhe (TH),
Zirkel 2, D-76128 Karlsruhe, Germany
Phone: +49 721 608 6411, Fax: +49 721 388097, bless@tm.uka.de

Abstract. Results from simulation models can be used to derive implications for real scenarios only when the model is designed very close to reality. Besides solving the question of how to generate data traffic in the simulation, one has to solve also the problem of how to generate larger realistic topologies. The latter are required in many cases to evaluate protocols and mechanisms in respect to scalability. Instead of using algorithms for generating Internet-like topologies artificially, it is possible to use sources of the real Autonomous System topology for generating the simulation topology. This paper describes a first approach of how to make use of such data for OMNeT++ simulations and presents first experiences.

1 Introduction

Results from simulation models can be used to derive implications for real scenarios only when the model is designed very close to reality. Network simulations for Internet-related protocols require several special considerations [2]. Besides solving the question of how to generate data traffic in the simulation [7], one has to solve also the problem of how to generate larger realistic topologies. The latter are required in many cases to evaluate protocols and mechanisms in respect to scalability.

The real Internet topology consists of at least two different hierarchy levels. The coarser first level consists of several interconnected *Autonomous Systems (ASs)*. An Autonomous System constitutes an administrative domain, usually operated by an Internet service provider (ISP). The second hierarchy level can be found within an AS. It comprises all the routers and their interconnections within an AS (strictly speaking this level can consist of intra-domain hierarchy levels, too, e. g. OSPF routing areas). This topology is usually neither propagated nor directly visible outside the AS.

Algorithms for generating Internet-like topologies have to obey some rules which have recently been formulated as several "power laws" [1,4] for the first hierarchy level of ASs. More detailed analysis of the Internet topology by different approaches can be found in [8].

Instead of using algorithms for generating Internet-like topologies artificially, it is possible to use sources of the real AS topology for generating the simulation topology. This paper describes a first approach of how to make use of such data for OMNeT++ simulations and presents first experiences.

2 Getting Sources of Topology Information

The AS topology level can be used for evaluating different aspects of inter-domain protocols, e. g. stability of the border gateway protocol BGP [3] or scalability of new signaling protocols for resource reservation [6]. Currently, there exist more than 12000 different ASs which have each a unique assigned AS number. One can distinguish several different AS types: a *stub AS* is only source or sink of data, whereas a *transit AS* also carries and forwards traffic that has neither its destination nor its origin in this AS. Stub ASs can be *single-homed* or *multi-homed*, i. e. having only one or more connections to different providers respectively. Furthermore, there may exist *pure transit ASs* or *mixed ASs*. The latter carry transit traffic and have also networks that are sources and sinks of traffic.

One source for getting information about the real Internet AS level topology are routing tables for inter-domain routing generated by the routing protocol BGP. A routing table constructed by BGP within an AS represents only a particular view of this AS, because BGP is a path vector protocol. The routing table contains a set of AS paths for each destination network prefix address (cf. fig. 1). Therefore, not all connections between other ASs are visible for this particular AS. However, there exists a "route-views" project [5] at the University of Oregon in order to enable service providers to check their own routing information. Currently, over 50 providers are contributing their routing information to a dedicated BGP router. BGP routing tables are archived automatically several times a day and are made available for download.

```
route-views.oregon-ix.net>show ip bgp
BGP table version is 694918, local router ID is 198.32.162.100
Status codes: s suppressed, d damped, h history, * valid, > best, i - internal
Origin codes: i - IGP, e - EGP, ? - incomplete
0120123456789012345601234567890123456789012345601234560123456
   Network          Next Hop            Metric LocPrf Weight Path
*  3.0.0.0          204.42.253.253                       0 267 1225 701 80 i
*                   129.250.0.1              6           0 2914 701 80 i
*                   129.250.0.3              0           0 2914 701 80 i
...
*  128.134.151.128/25 195.66.225.254                       0 5459 3549 4766 9526 i
```

Fig. 1. A small excerpt from a BGP table dump

In comparison to BGP tables other sources of AS topology information, e. g. gathered by router-level path traces, show usually a smaller degree of coverage. Therefore, a small program written in Perl was used to produce a graph representation for the AS topology from the BGP routing table data archived from route-views. The BGP routing table data is usually dumped in a human readable ASCII format generated by a CISCO router as output from the command sequence `show ip bgp`. The routing table is quite huge in this representation: it comprises more than 360 MB in its uncompressed form.

The script also analyzes the type of ASs and creates a topology data file which is suitable for input by OMNeT++ simulations. When constructing the AS graph, an important simplification is made: it is assumed that links are bidirectional, so the graph is *undirected*. Furthermore, due to the fact that the assigned AS numbers are not always continuous, the real AS numbers are mapped to a continuous number space of unsigned integers starting at 1.

Source	Number ASs (transit, mixed, stub)	Conn-ections	Net-works	Paths	Max. path-length
route-views.oregon-ix.net (18.8.2001)	11621 (85,2637,8899)	26963	112276	4239791	31
route-views.oregon-ix.net (18.8.2000)	8326 (77,1894,6355)	18857	93513	1772139	20
route-server.ip.att.net (18.8.2000)	8221 (52,1107,7062)	12710	84116	1524708	17
route-server.cerf.net (18.8.2000)	8223 (47,1057,7119)	12382	83942	331700	17

Table 1. Characteristics of Autonomous Systems in the Internet

3 Implementation in OMNeT++

The objective of the now discussed OMNeT++ simulation was to evaluate a new inter-domain signaling protocol for resource reservation. In this particular case, it was assumed that there is only one signaling entity within each AS. Signaling messages were generated and consumed by hosts located in non-pure transit ASs. Furthermore, each signaling entity has to store a complete routing table to all other signaling entities. It was assumed that the routing tables are static, i. e. no link failures between ASs were simulated. Nevertheless, this implies a huge amount of memory usage for routing table data.

The design process in OMNeT++ was as follows: no module uses `activity()`, only `handleMessage()` was used in order to avoid problems with stack memory consumption when having several thousand modules. Instead of generating NED code from the topology data, OMNeT++'s feature for dynamic module creation was used (it is described in the OMNeT++-manual). If you look at output from the `nedc` compiler you see many repeated calls similar to those caused by unrolling loops. Therefore, it is more efficient to directly create the modules within OMNeT++ dynamically.

The OMNeT++ network description file contains only one simple module which reads in the topology data and creates all further modules dynamically during its `initialize()` call. Its `handleMessage()` method can be left empty. The dynamically created modules are subsequently initialized automatically by the OMNeT++

kernel. A topology file has the following format (assuming that total number of ASs is N):

```
N
asnum1 (type,outdegree)
...
asnumN (type,outdegree)
asnum1 : ASa ASb ASc .... ASz ;
...
asnumN : ASv ASw ASx .... ASr ;
```

The total number N of nodes in the graph is given in the first line of the topology data file. This is useful in order to know how many times one has to loop over the lines in the file, and, to temporarily store the actual module pointers in an array. Subsequently, a list of all nodes follows, describing its type (1 = single-homed, 2 = multi-homed, 3 = mixed, 4 = pure transit) and the outdegree, i. e. the number of outgoing links (or gates in OMNeT++ terminology). While parsing this list, the necessary modules are created dynamically (using `cModuleType::create()`) with their respective incoming and outgoing gates (using `cModule::setGateSize()`). The second list consists of an adjacency list which describes the interconnections between all ASs. Therefore, after the modules have been created, all necessary connections between the modules are established (using `connect()`). Modules for hosts are additionally generated and connected for each AS that is not a pure transit AS. A topology data file for the Internet topology from August 18th 2001 needs approximately 500 kB of file space.

A very simple 5 node topology with a pure transit AS in the middle is described by the following topology file:

```
5
1 (1,1)
2 (3,2)
3 (4,2)
4 (3,2)
5 (1,1)
1 : 2 ;
2 : 1 3 ;
3 : 2 4 ;
4 : 3 5 ;
5 : 4 ;
```

4 Experiences

The previously mentioned simulation model was used to run a simulation under Linux with an Internet topology from march 2000 consisting of 7183 ASs (53 transit, 1053 mixed and 6077 stub ASs). The routing tables were calculated by using the `cTopology` class (using `unweightedSingleShortestPathsTo()`) for each AS and dumped into a file in order to save calculation time when doing several simulation runs. The file size of all routing tables is around 738 MB in this case. The actual memory consumption is considerably higher because each routing tables entry needs additional pointers (each consuming 4 Bytes of memory). The current Linux kernel 2.4 limits the memory size of a user space process to 3 GB on Intel platforms so newer and current AS topologies could not be used yet. After simulation initialization the memory usage was around 2.2 GB. As hardware platform a two processor 1 GHz Pentium III with 4 GB RAM was used.

Creating and connecting all the modules (more than 14000, counting signaling nodes and hosts) from the topology data file is very fast (only 2 seconds), whereas initializing them requires some tens of seconds. The routing table calculation was still reasonably fast (around 20 minutes), but reading the precalculated routing tables from a file reduces the setup time considerably (down to 3 minutes). However, during all the time the OMNeT++ simulation kernel was stable.

5 Conclusion and Outlook

OMNeT++ is a very efficient simulation environment when doing large simulations with several thousands of modules. This makes it possible to use “real” large scale topologies for network simulations. It is planned to run simulations with more current AS Internet topologies on 64-bit platforms in order to break the 3 GB process memory limit on Intel Linux platforms. Furthermore, it is planned to generalize the presented first approach for topology data file input and to integrate it into the OMNeT++ kernel.

References

1. M. Faloutsos, P. Faloutsos, and C. Faloutsos. On Power-Law Relationships of the Internet Topology. In *Proceedings of the ACM symposium on Communications architectures & protocols*. ACM, 1999. SIGCOMM 1999.
2. S. Floyd and V. Paxson. Difficulties in Simulating the Internet. *IEEE/ACM Transactions on Networking*, 9(4):392–403, Aug. 2001.
3. S. Halabi and D. McPherson. *Internet Routing Architectures, Second Edition*. Cisco Press, 201 West 103rd Street, Indianapolis, IN 46290, USA, 2 edition, 2000. ISBN 1-57870-233-X.
4. A. Medina, I. Matta, and J. Byers. On the Origin of Power Laws in Internet Topologies. *ACM Computer Communications Review*, 30(2), 2000.
5. D. Meyer. Route views project page. `http://www.antc.uoregon.edu/route-views/`, Sept. 2000.
6. Next steps in signaling (nsis) charter. `http://www.ietf.org/html.charters/nsis-charter.html`, Dec. 2001.
7. K. Park and W. Willinger, editors. *Self-Similar Network Traffic and Performance Evaluation*. Wiley, 2000.
8. D. Vukadinović, P. Huang, and T. Erlebach. A Spectral Analysis of the Internet Topology. Technical Report ETH TIK-Nr. 118, Inter-Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, July 2001. erhltlich unter `http://www.tik.ee.ethz.ch/˜huang/publication/nls-tr.pdf`.

Session 2:
Parallel and
Distributed Simulation

Parallel Simulation with OMNeT++ using the Statistical Synchronization Method

Gábor Lencse

Széchenyi István University, Department of Telecommunications
Hédervári u. 3. H-9026 Győr, Hungary
Lencse@szif.hu
http://www.hit.bme.hu/phd/lencse

Abstract. The synchronization methods for parallel discrete event simulation (PDES) are compared. The original Statistical Synchronization Method (SSM) and its time driven version (SSM-T) are described. The PDES support of OMNeT++ is explained. A case study is given about the parallel simulation with OMNeT++: the simulation of two interconnected FDDI rings executed on two processors. The presentation contains also a demonstration of this parallel simulation at the workshop.

Introduction

Parallel discrete event simulation (PDES) could be an important tool for the performance analysis of communication systems. The precise simulation of these large and complex systems would really benefit from computing power of parallel systems. However PDES is a difficult problem. Fujimoto [1] describes the well-known synchronization methods for PDES. The aim of the synchronization methods is to deal with the problem of causality.

The conservative method avoids the causality errors, by executing only *safe* events. (For a segment S_1 an event E_1 with timestamp T_1 is safe if S_1 can determine that it is impossible for it to receive an event with timestamp smaller than T_1.) Mechanisms were developed for either deadlock avoidance or deadlock detection and recovery. The main problem of the conservative method is that it can produce good speed-up only for systems that meet special criteria, otherwise the method cannot exploit the possible parallelism.

The optimistic method allows the causality errors. Then it detects and recovers from them. There are several variants of the optimistic method offering different solutions. One of the most popular is Time Warp. It performs periodic state saving. If the state-saving overhead is large it may seriously degrade the performance of the simulation. Dynamic memory allocation of the processes causes further complications.

Both of the synchronization methods mentioned above have their own limitations and require quite much support from the simulation kernel. The Statistical Synchronization Method [2] is a promising alternative. SSM does not exchange individual messages between the segments but rather the statistical characteristics of the message flow. Actual messages are regenerated from the statistics at the receiving side. It has numerous advantages over the two other methods.

This paper first briefly describes the Statistical Synchronization Method, than the parallel simulation functionalities of OMNeT++ and finally a case study: parallel simulation of two interconnected FDDI rings.

1 The Statistical Synchronization Method

In its original form, SSM was invented by György Pongor [2] at the Lappeenranta University of Technology, Finland. It was further developed (as SSM-T) by Gábor Lencse [3] at the Technical University of Budapest, Hungary.

1.1 The Original SSM

Similarly to other parallel discrete event simulation methods, the model to be simulated – which is more or less a precise representation of a real system – is divided into segments, where the segments usually describe the behavior of functional units of the real system. The communication of the segments can be represented by sending and receiving various messages. For SSM, each segment is equipped with one or more input and output interface. The messages generated in a given segment and processed in another segment are not transmitted there but the *output interfaces* (OIF) collect statistical data of them. *The input interfaces* (IIF) generate messages for the segments according to the statistical characteristics of the messages collected by the proper output interfaces. (See *Fig. 1.)*

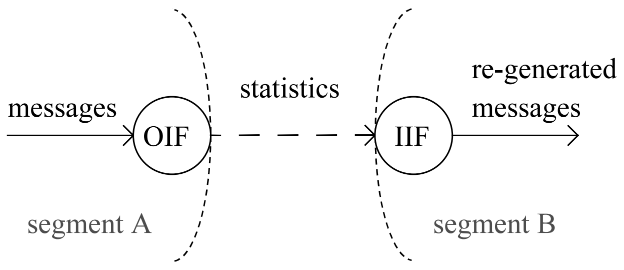


Fig. 1. An OIF – IIF pair

The segments with their input and output interfaces can be simulated separately on separate processors, giving statistically correct results. The events in one segment do not have the same effect in other segments as in the original model, so the results collected during SSM are not exact. The precision depends on the partitioning of the model, on the accuracy of statistics collection and regeneration, and on the frequency of the statistics exchange among the processors.

SSM has the following advantages compared to the other PDES synchronization methods:

- requires less network bandwidth
- tolerates communication delay better
- can be easily implemented
- requires less support from the simulation kernel
- may produce better speed-up

1.2 The Definition of SSM-T

In its original form, SSM was applicable for the analysis of steady state behavior of systems. In another paper [4] Dr Pongor emphasizes the advantages of the fact that the local virtual time (LVT) of the segments may be completely different, this feature may result in automatic importance sampling and super optimal speed-up.

However, an approximate synchronism of the LVTs of the segments is often desirable. For example the load of communication systems is not constant, but changes during the day according to a hat-like curve. A simulator should be able to follow this behavior. At the Department of Telecommunications, Technical University of Budapest, we further developed SSM to have this property. This version is distinguished as SSM-T, the time driven version of SSM. We also completed OMNeT++ with the necessary functionality for parallel execution.

The basic idea of SSM-T is very simple. Let the execution of the segments run independently in the majority of time facilitating good speed-up and let the LVTs of the segments meet at certain points of time ensuring an approximate synchronism.

Loose synchronization between segment A and segment B is defined formally as follows:

Let $t_1, t_2, t_3, \dots t_i, \dots t_n$ be synchronization points of time. Let t_A and t_B denote the LVTs of segment A and segment B, respectively. Segment A and segment B are loosely synchronized if:

$$((t_A < t_i) \Rightarrow (t_B \le t_i)) \wedge ((t_A > t_i) \Rightarrow (t_B \ge t_i)),\ \ i = 1, 2, \dots , n. \qquad \mathbf{(1)}$$

The loose synchronization of the two segments means that none of them may leave any synchronization point of time (t_i) until the other one reached it.

At the t_i synchronization point of time segments A and B may exchange the statistics they have collected before t_i and they may use the new ones in the $[t_i, t_{i+1}]$ time interval. With the appropriate choice of the t_i synchronization point of time, it is ensured that the effect of a change in segment A in t_x will reach segment B earliest at t_x and latest at $t_x+\Delta t$. In the simplest case, we use loose synchronization where $t_i=i*UI$, UI being the update interval.

What does this method require from the simulation kernel? Let us consider the following example. If *segment A* wants to send a message to *segment B* at LVT t_i then segment A should ask the simulator not to let the LVT of segment B pass t_i until segment B receives a message from segment A. (For simplicity and clarity we consider only one direction of communication as the other direction can be handled in the same way. If both segments ask for synchronization for the same LVT, then the LVTs of the two segments will really meet.) As for the implementation, if segment A sends a *synchronization point request* to segment B for LVT t_i than segment B schedules a self message (a special event, that will be processed by the simulation kernel itself) for LVT t_i. If the statistics package from segment A arrives to segment B at LVT $t_x<t_i$ than the message carrying the statistics is simply scheduled to t_i. If the statistics package from segment A to segment B does not arrive until t_i, segment B processes the self message and suspends processing of any further events until the statistics package arrives. Fig. 2. shows examples for both cases.

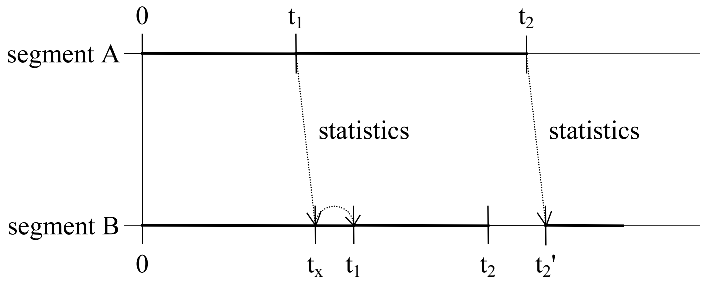


Fig. 2. The operation of SSM-T. The thin horizontal lines show the wall-clock (real) time of the processors executing the segments and the thick lines are the virtual times of the segments

1.3 Further Investigations on SSM-T

Because of space limitations we cannot cite all the results on SSM-T, instead we briefly summarize the topics covered in previous papers. All the cited papers can be downloaded from the author's web page.

Accuracy of SSM is dealt with in [5]. Different statistics collection algorithms were examined concerning their resource requirements and accuracy.

The applicability criteria of SSM are given in [6] together with examples when the method can or cannot be used.

The Statistics Exchange Control Algorithm [7] is responsible for determining the appropriate virtual time for the next statistics exchange. Its task is not at all trivial, as the required accuracy gives the number of observations required for the statistics, and the statistics exchange control algorithm must meet the correct virtual time. If the prediction fails, the synchronization point is deleted and the statistics collection must be continued. The effects of the different kinds of deviations from the optimal case were examined.

2 Implementing Parallelism in OMNeT++

OMNeT++ can run parallel on different types of hosts and/or operating systems that support PVM.

2.1 Parallel Topology Description

OMNET++ parallel support includes the parallel topology description, a flexible method, that enables the user to describe the partitioning of the model. When defining a module type, a compound module (besides the formal parameter list and the submodule list) may have a *formal machine list* introduced by the keyword `machines`. When defining the inside structure of the compound module, the user can express onto which machine (from the formal machine list) he wants to place the submodules: the keyword `on` is followed by the machine. Let us see an example:

```
module Network
     machines:
          host4netA, host4netB;
     submodules:
          netA: subnetwork;
               on: host4netA;
          netB: subnetwork;
               on: host4netB;
     connections:
          netA.out --> netB.in;
          netB.out --> netA.in;
endmodule
```

When building the network topology, the simulation kernel evaluates the machine parameters and places the modules into a single segment (process) per host. Especially for testing/debugging purposes it is possible to place all the segments in separate processes onto one host, SINGLE_HOST has to be defined in the `pvmmod.cc` source file.

2.2 Communication Between the Segments

In OMNeT++, the modules communicate with each other by sending and receiving various messages. The messages may contain arbitrarily complex data structures. Basically, the user does not have to take care if the modules communicating with each other are placed into the same segment or into different ones. If communication occurs between two modules that are in different segments, OMNeT++ packs all the standard data structures – that are included in the message – into PVM messages and carries them safely to the destination. Of course there are some natural restrictions:

1. Pointers became meaningless if they are transferred from one process to another

2. The conversion of the different architectures is done interpreting data as their types are defined. For example the order of the 4 bytes of an integer is reversed if it is transferred between a little-endian and a big-endian computer. The dirty trick of accessing the 4 bytes of this integer as `char` type variables will not work well.
3. The range of the possible values of data types may differ. For example the storage size of a variable of C++ type `long` is 4 or 8 bytes on a 32 or 64 bit architecture computer.

2.3 Synchronization Between the Segments

OMNeT++ provides a very simple mechanism for the inter-segment synchronization. The user may send a `synchpoint(time)` from one segment to another. The target segment's LVT may not pass `time` (and its execution is suspended if it has no more events with less or equal timestamp than `time`) until a message from the source segment arrives. The first synchronization points are sent at the beginning of the simulation and the user must take care to send the next synchronization point always *before* he sends the expected message that deletes the actual synchronization point. This mechanism can be used for either conservative or statistical synchronization. The latter is supported by different statistics collecting classes, such as histograms, P^2, K-split [8].

3 Parallel Simulation of Two Interconnected FDDI Rings

We would like to demonstrate the parallel simulation with OMNeT++ on the example of an FDDI network. In 1996, the backbone of the Technical University of Budapest consisted of two rings: The Northern Ring was a university-wide network and consisted of 15 FDDI stations interconnected by 5 wiring concentrators. The Southern Ring was the backbone of the Faculty of Electrical Engineering and Informatics, and being a smaller ring consisted only of 7 FDDI stations. Figure 3. shows the topology of the network. The two rings are interconnected by the *bmecisco7* router. The topology of the network and the cable lengths were taken from the real system.

The load used in the simulation model came from measurements taken on the real rings. By using a protocol analyzer, the first 32 octets of all the packets were copied from the ring and the packet lengths, arrival times, as well as the addresses of the source and destination stations were stored.

The natural segmentation of the network is to place each ring into its own segment. The bmecisco7 router is a part of both rings and is cut into two in the middle. The statistical interfaces are inserted between the two rings, each segment has one input and one output interface, as the data flow between the segments is bi-directional. The NeD description of the network looks as follows (the parameters were removed to save space):

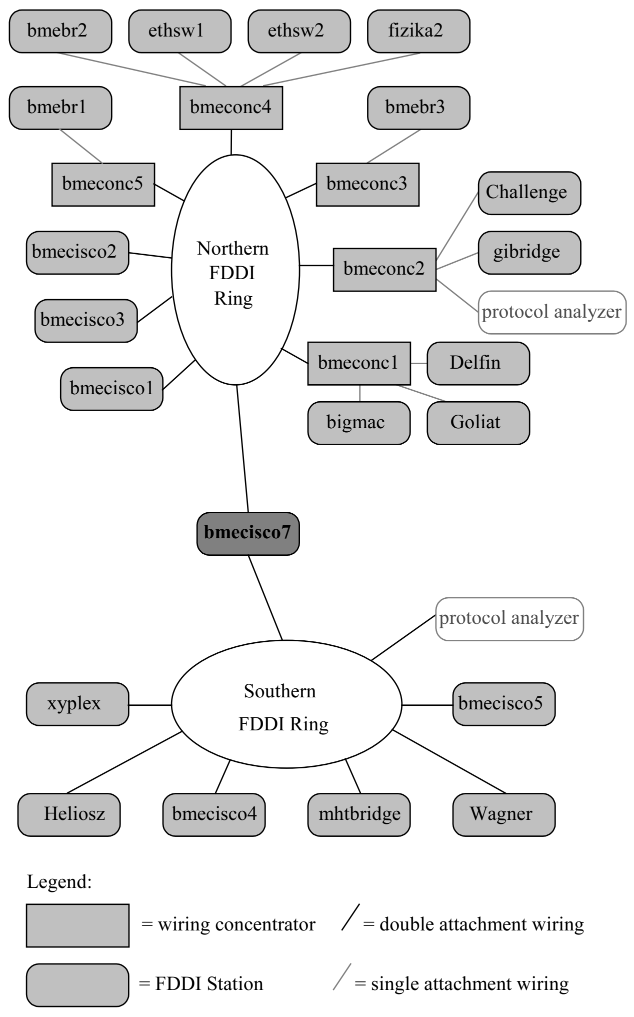
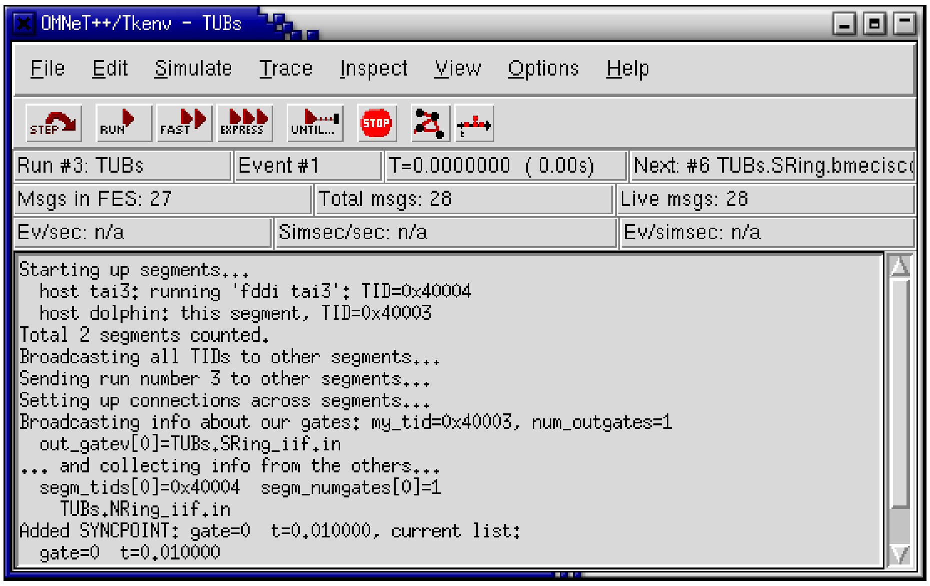


Fig. 3. The FDDI backbone of the Technical University of Budapest in 1996

```
module TUB_SSM
  machines:
    NR_host, SR_host;
  submodules:
    NRing: TUBNRing;
      on: NR_host;
    SRing: TUBSRing;
      on: SR_host;
    NRing_oif: SSM_OIF_type like SSM_OIF;
      on: NR_host;
    NRing_iif: SSM_IIF;
```

```
        on: NR_host;
      SRing_oif: SSM_OIF_type like SSM_OIF;
        on: SR_host;
      SRing_iif: SSM_IIF;
        on: SR_host;
    connections:
      NRing.out --> NRing_oif.in;
      NRing_oif.out --> delay 0.05 us --> SRing_iif.in;
      SRing_iif.out --> SRing.in;
      SRing.out --> SRing_oif.in;
      SRing_oif.out --> delay 0.05 us --> NRing_iif.in;
      NRing_iif.out --> NRing.in;
  endmodule
```



```
Run #3: TUBs | Event #1 | T=0.0000000  ( 0.00s) | Next: #6 TUBs.SRing.bmecisc
Msgs in FES: 27 | Total msgs: 28 | Live msgs: 28
Ev/sec: n/a | Simsec/sec: n/a | Ev/simsec: n/a

Starting up segments...
  host tai3: running 'fddi tai3': TID=0x40004
  host dolphin: this segment, TID=0x40003
Total 2 segments counted.
Broadcasting all TIDs to other segments...
Sending run number 3 to other segments...
Setting up connections across segments...
Broadcasting info about our gates: my_tid=0x40003, num_outgates=1
  out_gatev[0]=TUBs.SRing_iif.in
... and collecting info from the others...
  segm_tids[0]=0x40004  segm_numgates[0]=1
    TUBs.NRing_iif.in
Added SYNCPOINT: gate=0  t=0.010000, current list:
  gate=0  t=0.010000
```

Fig. 4. The main window of OMNeT++

The demonstration of the parallel simulation runs on two notebooks, interconnected via 10BaseT Ethernet. The PVM version is 3.4.4, OMNeT++ the currently available 2.1 distribution [9]. The complete FDDI model can be found among the samples. The OMNeT++ manual gives a detailed description about how to configure PVM, and about getting the FDDI model run parallel. Despite the fact, that the FDDI simulation was written in 1997, before the public release of OMNeT++ the code still compiles and even works. (Thanks to András Varga, who always updated it when the simulator was changed.) For those, who would like to repeat the experiment: the above statements are true only if SINGLE_HOST is *not* defined. If it is defined the

2.1 distribution `src/sim/pvm/pvmmod.cc` file will not compile, but needs about 10 minutes hacking… Bugfix is planned to be provided soon.

4 Summary

We have shown, that SSM-T is a good solution for PDES synchronization. All the necessary functionality is already included in OMNeT++. So now we are looking for volunteers, who test the method for real life applications.

References

1. Fujimoto, R. M.: "Parallel Discrete Event Simulation". *Communications of the ACM* 33, (1990.) no 10, pp. 31-53
2. Pongor, Gy.: "Statistical Synchronization: a Different Approach of Parallel Discrete Event Simulation". *Proceedings of the 1992 European Simulation Symposium (ESS'92)* (Dresden, Germany. Nov. 5-8) SCS Europe, pp. 125-129.
3. Lencse, G.: "Efficient Parallel Simulation with the Statistical Synchronization Method" *Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation (CNDS'98)* (San Diego, CA. Jan. 11-14). SCS International, pp. 3-8.
4. Pongor, Gy.: "Multiple Virtual Times in Parallel Discrete Event Simulation". *Proceedings of the Parallel Processing Workshop* (Technical Univ. of Budapest, Budapest, Hungary, Febr. 10-11, 1994.)
5. Lencse, G.: "Statistics Collection for the Statistical Synchronisation Method" *Proceedings of the 1998 European Simulation Symposium (ESS'98)* (Nottingham, UK. Oct. 26-28). SCS Europe, pp. 46-51.
6. Lencse, G.: "Applicability Criteria of the Statistical Synchronization Method" *Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation (CNDS'99)* (San Francisco, CA. Jan. 17-20). SCS International, pp. 159-164.
7. Lencse, G.: "Design Criterion for the Statistics Exchange Control Algorithm used in the Statistical Synchronization Method" *Proceedings of the Advanced Simulation Technologies Conference (ASTC 1999)* part of the 32nd Annual Simulation Symposium (San Diego, CA. April 11-15) SCS International, pp. 138-144.
8. Varga, A.: "K-split – On-Line Density Estimation for Simulation Result Collection". *Proceedings of the 1998 European Simulation Symposium (ESS 98)* (Nottingham, UK. Oct. 26-28,.). SCS Europe, 41-45.
9. Varga, A.: "OMNeT++ Discrete Event Simulation System" (2001) http://www.hit.bme.hu/phd/vargaa/omnetpp.htm

A Networked Remote Simulation Architecture and its Remote OMNeT++ Implementation

Erdei, Márk

Department of Telecommunications
Budapest University of Technology and Economics
merdei@hit.bme.hu

Sója, Katalin

Department of Telecommunications
Budapest University of Technology and Economics
ksoja@hit.bme.hu

Wagner, Ambrus

Department of Telecommunications
Budapest University of Technology and Economics
ambi@hit.bme.hu

Simulation is a resource-intensive task. It demands both computational power and large storage capacity. In this paper we present an architecture that provides a client-server distributed environment facilitating effective sharing of resources. In a simulation system, three function groups can be clearly separated: control, execution, and storage of models and results. Each of these functions requires a specialised resource profile. The benefit of separating the three function groups is that the costly simulation hardware becomes accessible to a wide audience and on the other hand user hardware requirements are kept to a minimum. The distributed simulation architecture we developed effectively supports the functional separation of simulation systems as outlined above. In this paper first the architecture is introduced, followed by the description of the various usage scenarios. Then the security system is outlined, and a brief overview of the implementation for the OMNeT++ simulator, called Remote OMNeT++, is given.

The Architecture

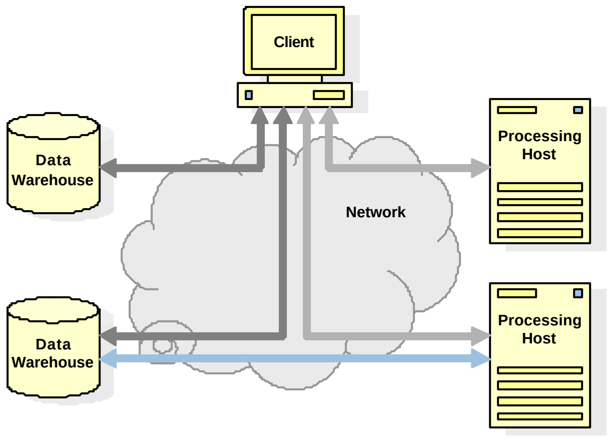
This architecture is designed to separate the different functions in a simulation system. This is necessary because the resource needs of each function are different. Three basic function groups can be distinguished:

- Control
- Simulation Execution
- Data Storage (models, results, etc.)

These functions are embodied in the three main components of our architecture respectively:

- Client
- Processing Host
- Data Warehouse

These components and their relationships are shown on the figure below:



The components of the architecture and their relationships

The Client allows the user to create and control simulation runs, manage models, and results. This program has a small footprint, this way the user does not need a high-performance workstation to use the simulation system.

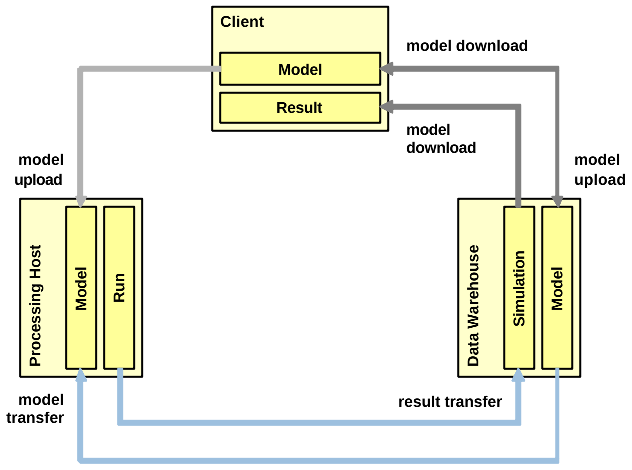
A Processing Host is typically an expensive machine, with enough processing power to execute multiple simulations at the same time. The number of simultaneous users is limited only by the capacity of this computer.

A Data Warehouse is a machine with large storage capacity to store simulation models and results. Through the use of Data Warehouses, models and results can be shared among users easily.

This architecture is a type of client-server architecture, in which the components are interconnected via a network, which can be an intranet, or the Internet, depending on the usage scenario.

On the figure below, the flow of data can be observed. There are four basic types of objects:

- Model: a simulation model (source code)
- Run: a running simulation
- Simulation: a finished simulation with a reference to the model and the results
- Result: a result of a finished simulation



Data flow between the components

In the following sections the components are described.

Client

The Client is the user's computer. Only a small software component has to be installed, so there are no special hardware requirements. The Client is basically a gateway to the rest of the simulation system. In the ideal case, neither simulation execution, nor data storage is located on the user's computer.

Using the Client software, the user can connect to the other components, that is, to the Processing Hosts and the Data Warehouses, and perform various actions, like data transfers and simulation executions.

Processing Host

The Processing Host is the high-performance simulation execution machine. It runs the simu lation engine to execute the simulation models. Processing Hosts do not have large storage capacities, therefore neither models, nor results are stored in Processing Hosts. Models are uploaded as needed, while results are automatically transferred to a Data Warehouse as soon as the simulation is completed.

Data Warehouse

The Data Warehouse is the storage component in the system. It stores simulation models for later use and simulation results for analysis. The models are uploaded by the users, and the results are transferred from the Processing Hosts upon simulation completion.

Usage Scenarios

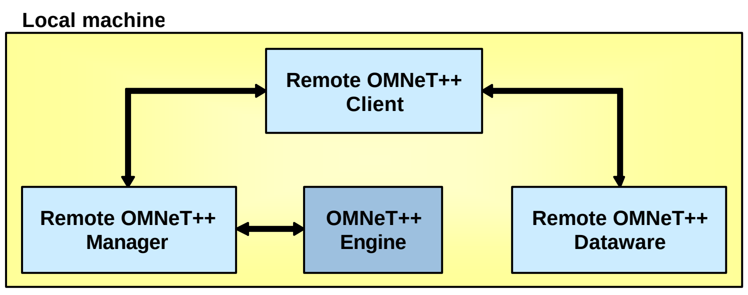
There are several different usage scenarios, based on the location of the individual components. The degree of spatial separation can be adjusted according to the application. This means, that in some cases, all three components should be on the same machine, while in other cases, they should be separated.

There are three main scenarios:

- Local machine
- Small research or educational laboratory
- Large, corporate laboratory

In the following sections the above three scenarios will be discussed.

Local machine

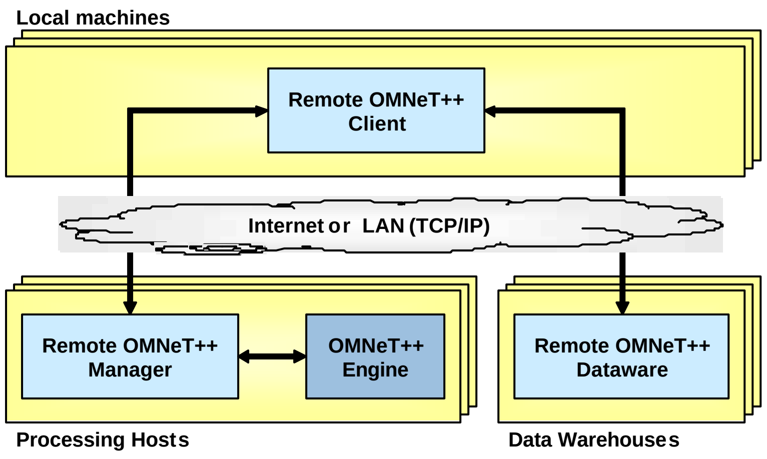


The local machine scenario

In this scenario, the three components are colocated on the user's computer. This is useful when a model is being developed. Clearly, it would not be effective to continuously upload the model to a Processing Host for testing, and then download the results. One advantage of the architecture is that the components are location transparent, that is, the same components can be used even if they reside on the same computer.

Obviously, if someone just wants to experiment with the system, this is the scenario he would use.

Small laboratory



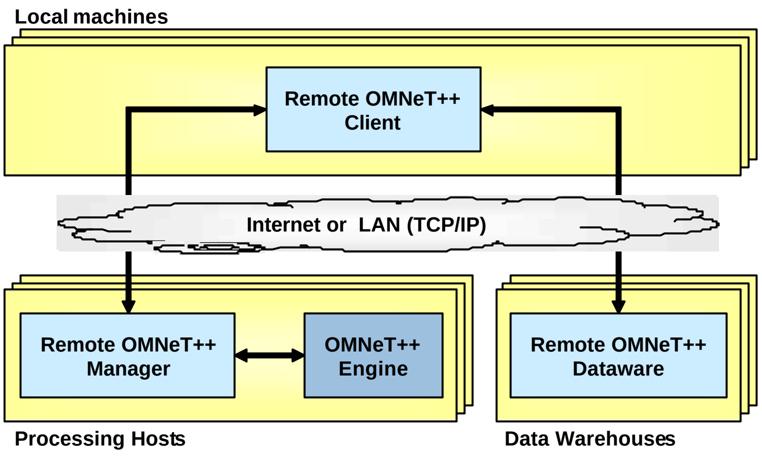
The small laboratory scenario

In this scenario, the Processing Host and the Data Warehouse is a single computer, with adequate processing power and storage capacity. Multiple Clients can then connect to this machine, execute simulations and download results.

This scenario is typical for a small laboratory, like a small research laboratory or a university laboratory, where there is no need for exceptional processing or storage capacity.

A university laboratory can benefit from the fact that models and results can be shared effectively among the users. The teacher uploads the model to the server, then the students download and maybe modify it. Next, they upload their models for simulation, and download the results for analysis.

Large laboratory



The large laboratory scenario

In this case all three components are located on different machines. In fact, there might be more than one Processing Host and Data Warehouse.

This scenario is appropriate for large laboratories, or distributed computing environments (like a Grid Computing environment), where there are a number of available resources, and the user selects the one that meets his simulation or storage needs.

A typical usage sequence

In this section, a typical sequence is described. The user wishes to simulate a model, that already exists on a Data Warehouse (A), and wants to store the results on a different Data Warehouse (B). The user connects to the resources over the Internet.

1. Login to the Data Warehouse A, and to the Processing Host.
2. Transfer the model from Data Warehouse A to the Processing Host. (The model is not transferred to the user's computer! With appropriate security settings (see next chapter) it is possible to allow a user to simulate a model, but not to download it. This is very useful if the model contains sensitive data.)

3. Create a simulation run, and specify Data Warehouse B as the destination for the results.
4. Start the simulation, and disconnect from the Internet.
5. Reconnect to check simulation progress.
6. When the simulation completed, download the results from Data Warehouse B.

Security System

Obviously, sharing data among different users is not without problems. In some cases it is not desirable for all users to have access to a model, results, or resources. User authentication must also be solved.

There are two main components of the security system in the architecture:

- User authentication
- Control of user actions

User authentication

User authentication means reliable identification of users. Also, user identification must be portable, to be able to attach rights to objects such as models.

Our solution is based on a public key encryption scheme. The first time a user connects to a server, he uploads his public key. The server ensures via a challenge-response method that the user is in possession of the private key, that is, the uploaded public key is indeed his own. Next, the user chooses a username and a password, that is specific to that particular server. However, the user is always identified by his public key. The username is used only to make logging into the server easier.

As an addition to the public key scheme, certificates could be used to verify a user's identity.

Control of user actions

There are many kinds of actions that can be controlled by the security system. Some of these are linked to servers, while others are linked to objects, like models or results. Server-based rights include the rights to upload a model to a server, download a model from a server, execute a simulation on that server, etc. Object-based rights include the right to simulate a model, view a set of results, etc.

In the case of server-based rights, the rights are stored on the server for each user. In the case of object-based rights, however, this information must be conveyed along with the object in question. That is, each object has an associated security information object, that contains information about the rights certain users have in connection with that particular object. This makes the objects transferrable, which is important when, for example, a model is moved from one Data Warehouse to another. In this context it is very important to be able to identify users uniquely, hence the need for the public key approach.

Rights attached to objects make the scheme mentioned in the previous chapter possible, where the user has no right to download a model, only to simulate it.

Implementation

In this section some implementation aspects are covered, concentrating on the user's perspective, omitting the details.

Our implementation was made for the OMNeT++ simulator, and is therefore called Remote OMNeT++. This system contains three main software components that correspond to the functions described earlier.

• Client:	Remote OMNeT++ Client
• Processing Host:	Remote OMNeT++ Manager
• Data Warehouse:	Remote OMNeT++ Dataware

All three components are implemented in Java, to make the system completely portable. The components communicate via a standard Java RMI interface. This makes it possible to develop proprietary components for the system, if necessary, for example to interface to a different simulator.

In the following sections, the individual components are described.

Remote OMNeT++ Client

This software component is a GUI-based component that is installed on the user's computer. It enables the user to connect to the server components of the system, that is, to Processing Hosts and Data Warehouses.

The user can start a simulation, continuously monitor its progress, pause, resume, stop, or abort the simulation. The user might even disconnect from the Processing Host while the simulation is running, and reconnect later to check its progress. This feature is especially useful for dial-up users.

Remote OMNeT++ Manager

This component is installed on Processing Hosts, and has three basic functions:

- compile uploaded models
- control the OMNeT++ simulation engine
- transfer the results to a preselected Data Warehouse

The first function enables users to upload the C and NED source code to the Processing Host, where it is automatically compiled into a simulation executable.

This executable is then started, and controlled via a Java RMI interface. This is made possible by a special OMNeT++ environment called RemoteEnv.

When the simulation is finished, the results are automatically transferred to a Data Warehouse for long-term storage, and later analysis.

Remote OMNeT++ Dataware

This is the software component for the Data Warehouses. It interfaces to some data storage mechanism to store user data. This might be the local file system, or a full-blown database management system, depending on the application.

Future Plans

There are two major plans for the future:

- CORBA interfaces
- Agent-based resource management

CORBA interfaces would make this system completely open-architecture, making it possible to develop specialised components in languages other than Java. This might be necessary, for example, when a high-performance simulation server does not have Java installed. In this case, a pure C implementation of some parts of the Remote OMNeT++ Manager is required.

Agent-based resource management helps users find appropriate resources in a distributed environment. This is closely related to Grid Computing, where processing power and storage capacity are provided and maintained by an organisation, which then makes these resources available to users. Fees are usually based on processor ticks and megabytes. If there are a large number of resources available, a software agent can be used to travel through the resource network and negotiate with the resource providers based on the preferences of the user.

Conclusion

The architecture presented in this paper solves the problem of shared use of valuable simulation equipment, and data. An implementation for the OMNeT++ simulator called Remote OMNeT++ has also been presented. Remote OMNeT++ is a small-footprint, scalable, platform-independent simulation environment for model developers, educational institutions, or even commercial applications.

Future plans include addition of CORBA interfaces for easy expansion, and agent-based resource management for Grid Computing environments.

References

Wagner, A., M. Erdei, *"Agent-Based Resource Management for Remote Simulation Systems and an Implementation for Remote OMNeT++"*, European Simulation Multiconference (ESM2001) , Prága, 2001.

Erdei, M., A. Wagner, K. Sója, M. Székely, "*A Networked Remote Simulation Architecture and its Remote OMNeT++ Implementation*", European Simulation Multiconference (ESM2001), Prága, 2001.

Erdei, M., K. Sója, A. Wagner, "*A distributed simulation architecture and its implementation: Remote OMNeT++*", Scientific Student Conference, Budapest University of Technology and Economics, Budapest, 2001.

Using Akaroa with Omnet++

Steffen Sroka[1] and Holger Karl[1]

Telecommunication Networks Group
Technical University Berlin
Berlin, Germany
`sroka@ft.tu-berlin.de, karl@ee.tu-berlin.de`
`http://www-tkn.ee.tu-berlin.de`

1 Introduction

When using discrete event simulation (DES) two main problems are: When is it ready to start collecting data (in order not to include initialisation effects) and when to terminate the simulation? One possible criterion is given by the confidence level, more precisely, by its width relative to the mean. But ex ante it is unknown how many observations have to be collected to achieve this level. Another problem is that DES can consume much time. Even with today's high speed processors simulation of modest complexity can take hours. AKAROA-2 [1] is a software packet that solves both problems. Akaroa was designed at the University of Canterbury in Christchurch, New Zealand and can be used free of charge for teaching and non-profit research activities. It is designed for running quantitative stochastic discrete event simulations on Unix multiprocessor systems or networks of heterogeneous Unix workstations. To speed up sequential simulation it uses *multiple replications in parallel (MRIP)*. This means that multiple instances of a sequential simulation program run on different processors. These instances run independently of one another and continuously send their observations to a central management process. This management process calculates from these observations an overall estimate of the mean value of each parameter. AKAROA-2 decides by a given confidence level and precision whether it has enough observations or not. When it judges that is has enough observations it halts the simulation. The simulation would be sped up approximately in proportion to the number of processors used and sometimes even more.

Currently these functionality is not present within OMNeT++ [2] and implementing these algorithms by hand is complicated and error-prone. Therefore it seems to be a good idea to integrate the AKAROA-2 capabilities into OMNeT++. Sequential simulation programs to be run under Akaroa must be written in either C or C++, or be capable of calling library routines written in C++. Therefore Omnet++ simulations are good candidates to use with Akaroa.

This document shows that OMNeT++ simulations can benefit from the capabilities of AKAROA-2 and describes a method to integrate the AKAROA-2 functions into existing simulations. Section 2 briefly describes how Akaroa works.

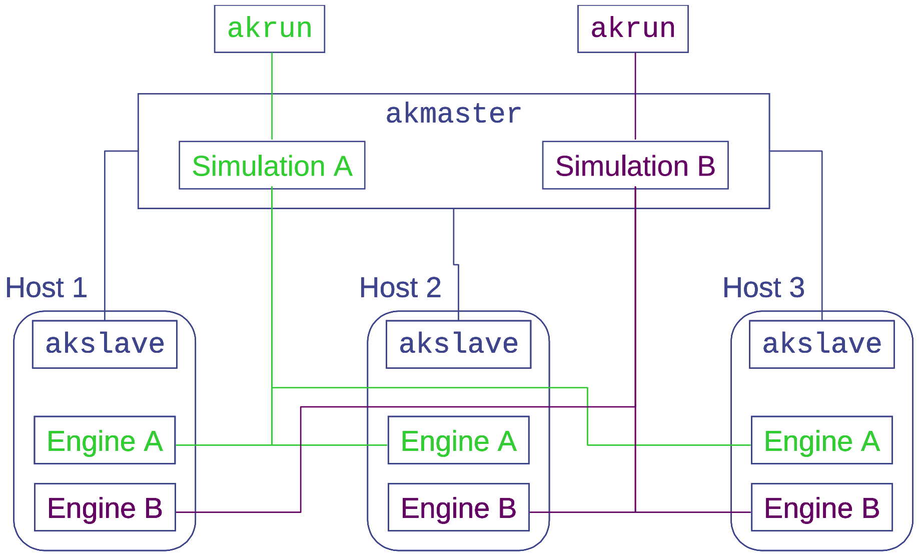


Fig. 1. Diagram of Akaroa running on 3 hosts

2 Running a simulation under Akaroa

This section is based on the manual of Akaroa. For a more detailed description see ref. [3] .

2.1 Parts of the Akaroa system

- Akmaster is the master process which coordinates all other processes in the Akaroa system.
- Akslave is a process that must run on each host that should be used for the simulation.
- Akrun instructs akmaster to launch the simulation on the requested number of hosts.

2.2 Starting up the Akaroa system

1. Start akmaster running in the background on some host.
2. On each host where you want to run a simulation engine, start akslave in the background.

In Fig.1 there is a scheme of a simulation running on 3 hosts.

2.3 Running a simulation

The akrun command starts a simulation, waits for it to complete, and writes a report of the results to the standard output. The basic usage of the akrun command is:

```
akrun -n num_hosts command [argument..]
```

where `command` is the name of the simulation you want to start. If the simulation is not in the search path you should use the full name to invoke. When akrun starts it reads the file `Akaroa` in the same directory. There are some variables that customize Akaroa. For more details see the Akaroa manual. Here is a little example which shows the output when starting the simulation uni on two hosts [3].

```
whio% akrun -n 2 uni
Simulation ID = 17
Simulation engine started: host = pukeko, pid = 23672
Simulation engine started: host = purau, pid = 434
Param    Estimate       Delta  Conf         Var   Count   Trans
    1    0.503476   0.0157353  0.95 4.42582e-05    1530     255
whio%
```

2.4 Shutting down the Akaroa system

For shutting down the Akaroa system it is only necessary to kill the akmaster process. Any other processes (akslave, akrun or simulation engines) attached to the akmaster process will be automatically terminated.

2.5 Additional parts of Akaroa

– Akadd adds machines to a running simulation.
– Akstat provides information about a running simulation.
– Akgui is a GUI written in Python, but actually does not work with some Linux installation.

3 Writing an OMNeT++ simulation for Akaroa

One basic virtue of Akaroa is that it is easy to adapt existing simulation programs to run under it. Only three steps have to be taken:

– Because Akaroa use MRIP it is indispensable that all simulation runs independently. When using the build-in RNGs from OMNeT++ every replication would work with the same stream of random numbers. Consequently the simulation should always obtain random numbers from the Akaroa system. It uses a Combined Multiple Recursive pseudorandom number generator (CMRG) with a period of approximately 2^{191} random numbers and provides a unique stream of random numbers for every simulation engine.

– The Akaroa system needs to know how many parameters are to observe. This information has to be transfered before the first observation is made. The function `AkDeclareParameters(n)` tells Akaroa that is has to handle `n` parameters.
– Finally all observations have to be transmitted to Akaroa. This is solved by the function `AkObservation(i,x)` that collects observations for parameter `i`.

3.1 Sub-classing AkOutVector from cOutVector

For an easy use of the Akaroa functionality a sub-classing from cOutVector seems to be ideal because this is the OMNeT++ class where such continuous observations are recorded. Thereby the last two steps from above can be hidden from the user. Then only replacing cOutVector by AkOutVector and the replacing of the random generators are required. Through sub-classing all functionality from cOutVector remains available. If only a single replication of a program is used, the replacing of RNGs is not required.

Listing 1 shows the declaration of AkOutVector. When calling the constructor of AkOutVector the constructor of cOutVector is called too. Thus all functionality of cOutVector is achieved. In order to tell Akaroa how many parameters have to be observed `veccount` is a static member. When the simulation records its first observation AkOutVector calls `AkDeclareParameters(veccount)` to tell Akaroa the number of parameters. Every time the simulation calls `record()` the class transfers the value to Akaroa and OMNeT++.

When running different instances of one simulation working on the same directory the names for storing data have to be different. Therefore the function `setOutFilename()` sets the name to `<hostname>.vec`.

3.2 Random Numbers

When running multiple replications of a simulation model in parallel, it is important that each simulation engine uses a unique stream of random numbers, independent of the streams used by other simulation engines. For this reason, if your simulation requires random numbers, you should *always* obtain them from the Akaroa system, so that Akaroa can coordinate the random number streams received by different simulation engines. The Akaroa library provides several distributions (see Listing 2).

3.3 Other Problems

If you use several instances of simple modules to form one estimate, you have to make AkOutVector* a static member of this simple modules class. This is a bit complicated because the constructor of cOutVector requires an other object. Listing 3 shows a possible solution.

Listing 1. Declaration of AkOutVector

```
#include "omnetpp.h"
#include <akaroa.H>
#include <akaroa/ak_message.H>

class AkOutVector : public cOutVector  {
   static bool Ak_declared;
   static long veccount;
   static bool hostfile;
   long Ak_id;
   void setOutFilename();
public:
        AkOutVector(const char *name=NULL) ;
        ~AkOutVector();
        virtual void record(double value);
};
```

Listing 2. Random distributions

```
#include <akaroa/distribution.H>

real Uniform(real a, real b);
long UniformInt(long n0, long n1);
long Binomial(long n, real p);
real Exponential(real m);
real Erlang(real m, real s);
real HyperExponential(real m, real s);
real Normal(real m, real s);
real LogNormal(real m, real s);
long Geometric(real m);
real HyperGeometric(real m, real s);
long Poisson(real m);
real Weibull(real alpha, real beta);
```

Listing 3. AkOutVector as static member

```
class ak_module : public cSimpleModule
{
        Module_Class_Members (ak_module, cSimpleModule, 16384);
public:
        virtual void activity ();
        virtual void finish();
        virtual void initialize();

        static AkOutVector* pParameter;
};
Define_Module( ak_module );


AkOutVector* ak_module::pParameter=NULL;

void ak_module::initialize()
{

if (pParameter==NULL)
        {
        pParameter=new AkOutVector("pParameter");
        };
}
```

4 Known Problems and Future Work

4.1 Terminating the Simulations

Currently Akaroa simply terminates the simulation engines when it has enough observations. The problem with this is that OMNeT++ does not call `finish()` when it receives such signals.

Another problem occurs when recording data within dynamically generated modules because the parameter has to be declared in advance to the Akaroa system. A solution would be a parameter placed in `omnet.ini` that contains the number of parameters to be observed.

5 Installing Akaroa under Linux

The installation of Akaroa under Linux needs some modifications of the Akaroa source files. A modified packet of Akaroa is available under `www-tkn.ee.tu-berlin.de/research/omnet-akaroa/`. For a detailed installation instruction read the read.me file in this folder.

References

1. K. Pawlikowski; D. McNickle; G. Ewing; R. Lee; J. Jeong. Project akaroa. Technical report, Department of Computer Science, University of Canterbury, Christchurch, New Zealand, www.cosc.canterbury.ac.nz/research/RG/.
2. Andreas Varga. Omnet++ eiscrete event simulation system. Technical report, Technical University of Budapest, www.hit.bme.hu/phd/vargaa/omnetpp/, 2001.
3. Greg Ewing; Krzysztof Pawlikowski; Donald McNickle. *Akaroa 2.6 User's Manual.* www.cosc.canterbury.ac.nz/research/RG, July 2000.

Session 3:
Simulating Wireless and Mobile Networks

A Discrete Model of the Mobile Radio Channel in OMNeT++

Dipl.-Ing. Timo Weiss

Universität Karlsruhe, Institut für Nachrichtentechnik, D-76128 Karlsruhe, Germany
`weiss@int.uni-karlsruhe.de`

Abstract. This paper presents a new approach in modelling the mobile radio channel which is especially suited for discrete event simulation. A multitude of parameters characterizing the wireless data transmission is taken into account i.e. several types of fading, delay and interference. The model delivers a realistic representation of the channel and the base band processing without demanding a large amount of processing power.
This is achieved by an abstraction from simulating every parasitic effect of the air medium and the base band in detail. Many of them can be merged to groups having the same impact on the bit error rate of the channel. Thus, a reduction of simulation complexity is obtained without sacrificing accuracy. The application of random process generators which are especially qualified for nonlinear time progession provokes a further speed enhancement. An elegant mathematical description of effects like cochannel interference and multiple access interference is introduced.
Further on, the implementation of the presented model in OMNeT++ is described. The specification of the model interface between the channel and the physical layer as well as between the physical layer and the data link layer is given. Finally, simulation results will be presented showing a good match between stream driven and discrete event simulation in terms of the first and second order statistical properties of the generated bit error process.

1 Introduction

In publications of recent years on simulation of higher protocol layers in mobile communication systems the special characteristics of mobile radio transmission is often taken into account insufficiently. In many cases simplified models or even plain AWGN-channels are employed.

Abstraction from reality is certainly necessary for computational reasons. However, the impact of many physical phenomena on functions of higher layers cannot always be neglected. For example, the time correlation of fading induced in the mobile radio channel plays an important role when it comes to the performance of MAC and DLC layer techniques.

But on the other hand we may not make the mistake of going into detail too deeply and simulate every bit with the accuracy of stream driven simulation tools. Thus, rare protocol events could not be investigated due to restricted computational power. Hence, a compromise is to be found between expenditure and accuracy. In the sequel, we describe a new approach providing more flexibility and closeness to reality and only requiring a reasonable amount of computational ressources.

For higher layers of the protocol stack only the bit and packet error processes of the channel are relevant. The description of the first and second order statistics of these processes and the influence of various channel and signal processing parameters is subject of this chapter.

1.1 Physical phenomena

First of all, important parameters and properties of the mobile radio channel are to be identified which are considered in the following.

- position and speed of each station
- Rice, Rayleigh, Gauss and lognormal fading
- free-space attenuation

– propagation and transmission delay
– external interference (MAI, CCI etc.)
– internal interference (ISI, ICI, sync mismatch etc.)
– additive white Gaussian noise

where external interference denotes the interaction between stations inside the propagation area. If only one link between two stations is considered (point-to-point) these interactions can be overcome by increasing the transmitter gain. Whereas internal interference denotes the distortion that is introduced within each link. This kind of interference is also present even in the absence of other users and cannot be mitigated by a higher gain which can be observed with ISI for instance.

1.2 Transmission model

In this section an abstracted channel model will be derived from which the important process of the instantaneous SNIR $\gamma(t)$ can be generated directly. This process $\gamma(t)$ represents the basis for the generation of the bit error process which is calculated in the physical layer (see next section).

Before the transmission model is described in detail some assumptions have to be made. First, the channel is assumed to be all-multiplicative i.e. the channel can be perfectly represented by its time-variant complex attenuation process $\underline{H}(t)$. Nevertheless, frequency selective properties of the mobile radio channel can also be embedded in this model as will be seen later on. Additionally, it is assumed that the receiver is able to demodulate perfectly coherently. Hence, the carrier phase is known and can be set to zero without loss of generality

$$\mathrm{Re}\{\underline{H}(t)\} = |\underline{H}(t)| \tag{1}$$
$$\mathrm{Im}\{\underline{H}(t)\} = 0. \tag{2}$$

For the sake of simplicity $\underline{H}(t)$ is now referred to as $H(t)$ and represents a real process. As it is shown later the assumption of perfect coherent demodulation does not preclude that synchronization and channel estimation errors can be taken into account as well. The channel process shall be normalized having the average power $E\{H^2\} = 1$. Note that $H(t)$ is a real process now as its phase is zero.

Furthermore, three noise processes are introduced representing the impact of the perturbations additive noise $\underline{N}_o(t)$, internal interference $\underline{N}_i(t)$ and external interference $\underline{N}_e(t)$ as mentioned above. The processes $\underline{N}_i(t)$ and $\underline{N}_o(t)$ can be considered stationary. Whereas, $\underline{N}_e(t)$ is generally nonstationary and depends on the number and the geometric positions of the participating stations and their transmission behavior. However, these parameters are always known to the simulation system.

As these parameters influencing $\underline{N}_e(t)$ are slowly changing $\underline{N}_e(t)$ is assumed to be stationary at least for small time intervals.

The average power of each noise process is defined as follows

$$\begin{aligned} E\{\underline{N}_0\underline{N}_0^*\} &= P_{N_0} \\ E\{\underline{N}_i\underline{N}_i^*\} &= P_{N_i} \\ E\{\underline{N}_e\underline{N}_e^*\} &= P_{N_e}. \end{aligned} \tag{3}$$

With all these definitions the transmission model can be given as shown in Fig. 1.

Where $\underline{s}(t)$ and $\underline{r}(t)$ denote the transmitted and the received signal, respectively, with $\underline{s}(t)$ having the average power $E\{\underline{SS}^*\} = P_S$. The amplitude gain $A(t)$ at the transmitter is time-variant in order to simulate power control and power saving mechanisms as well.

Now it is straight forward to derive the instantaneous SNIR $\gamma(t)$ from Fig. 1 as the ratio of wanted signal power to noise signal power in $\underline{r}(t)$.

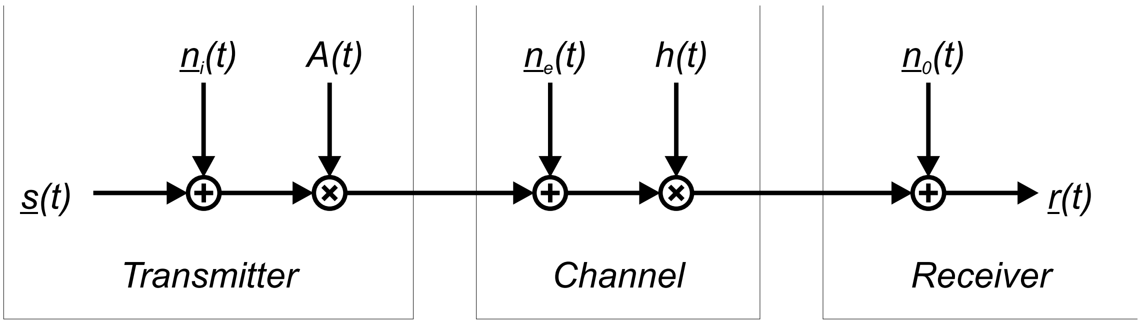


Fig. 1. The transmission model

$$\begin{aligned}\gamma(t) &= \frac{E\{\underline{S}\underline{S}^*\}A(t)^2h(t)^2}{(A(t)^2E\{\underline{N}_i\underline{N}_i^*\}+E\{\underline{N}_e\underline{N}_e^*\})h(t)^2+E\{\underline{N}_0\underline{N}_0^*\}} \\ &= \frac{P_S A(t)^2h(t)^2}{(A(t)^2P_{N_i}+P_{N_e})h(t)^2+P_{N_0}}\end{aligned} \tag{4}$$

Taking eq. (4) into account, neglecting all interferences $P_{N_i} = P_{N_e} = 0$ and assuming $h(t)$ and $A(t)$ to be time constant $\gamma(t)$ simplifies to

$$\gamma(t) = \frac{P_S A^2}{P_{N_0}} = \gamma_{AWGN}. \tag{5}$$

If we keep $h(t)$ time variant and Rayleigh distributed, this results in a flat Rayleigh channel

$$\gamma(t) = h(t)^2\frac{P_S A^2}{P_{N_0}} = h(t)^2\gamma_{AWGN}. \tag{6}$$

With $P_{N_e} = 0$ and $P_{N_o} \rightarrow 0$ (thus $\gamma_{AWGN} \rightarrow \infty$) we obtain

$$\gamma(t) = \frac{P_S A^2h(t)^2}{P_{N_i}A^2h(t)^2} = \frac{P_S}{P_{N_i}} = \gamma_{floor}. \tag{7}$$

From eq. (7) it is obvious that the internal interferences cause a SNIR-floor γ_{floor} provoking wrong decisions at the receiver even in the absence of AWGN. This error floor behavior is well known from many simulations and measurements of real systems.

Hence, eq. (4) can be considered a generalization of the special cases shown above (eq. (5) - eq. (7)), thus covering a multitude of phenomena in mobile radio transmission. How the necessary noise powers can be acquired in a practical application is subject of section 2. Next, we want to turn to a new technique for the realization of the process $H(t)$ which is especially suited for discrete event simulation systems.

1.3 Modeling the channel process

If one does not want to rely on a simple modeling of the channel process $H(t)$ using analytical Rayleigh and Rice models [Cla68][Jak93] one can skip to more sophisticated deterministic channel models. In the literature one finds two basic approaches for the implementation of a colored Gaussian noise (CGN) generator which represents a fundamental element of a channel simulator. There is the filter method and the Rice method.

In stream driven simulations the filter method as depicted in Fig. 2 is most common. Here, a CGN process is generated by filtering a white Gaussian noise (WGN) process with a linear time nonvariant filter having a given frequency response $G(f)$. If the input process is $\mathcal{N}(0,1)$-distributed the output signal of the filter is Gauss distributed as well with the power spectral density $|G(f)|^2$

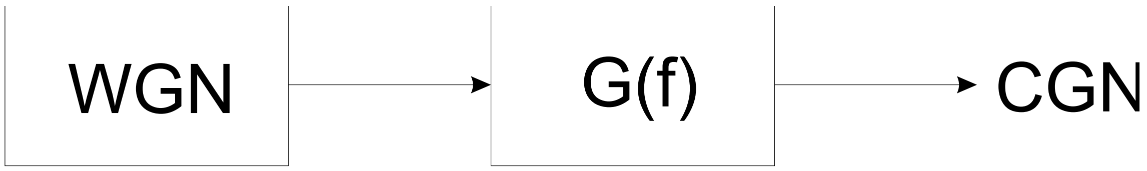


Fig. 2. The filter method

However, from a discrete event simulation point of view there is one substantial drawback no matter what kind of linear filter is employed: discrete leaps in time of different length are not possible. So if two events are separated by ΔT i.e. the end of a data transmission and the start of a new one then the filter has to be running throughout the entire time period ΔT between these two events although the state of the channel is irrelevant. This leads to an unnecessary computational overhead.

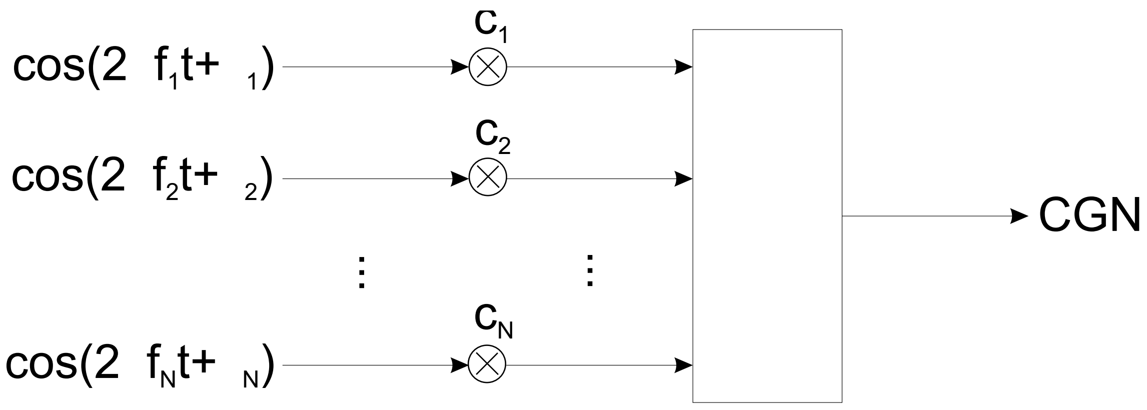


Fig. 3. The Rice method

A remedy can be found by generating colored Gaussian processes employing the Rice method. The main principle of the Rice method [Ric45] is shown in Fig. 3. It is based on the weighted superposition of an infinite number of weighted harmonic functions having different frequencies and random phases. Thus, a colored Gaussian process can be described by

$$h(t) = \sum_{n=0}^{\infty} c_n \cos(2\pi f_n t + \varphi_n). \tag{8}$$

However, for computational reasons one can only consider a limited number of superimposed waves in practice. As one can see from eq. (8), $h(t)$ is determined at each point in time making it possible to allow nonlinear time progression. This property makes the Rice method very attractive to discrete event systems.

Thereby, the phases φ_n can be chosen randomly and independently in the interval $[0, 2\pi)$. In this work the weight factors c_n were calculated using the method of equal areas due to its small computational expenditure and its quasi-nonperiodic correlation properties. The reader may refer to [Pae94] for details.

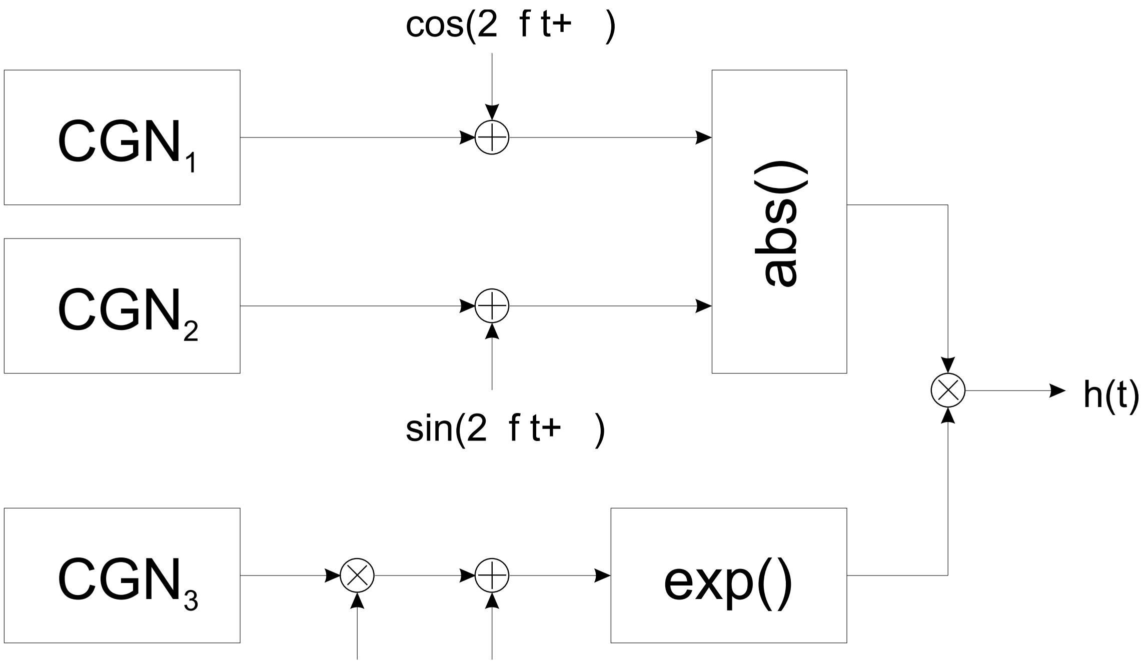


Fig. 4. Extended Suzuki process of type I

By combining several independent colored Gaussian noise generators as depicted in Fig. 3, Rayleigh, Rice and the more sophisticated extended Suzuki processes of type I [PKL98] and type II [PKL97] can be created. Exemplarily, a block diagram of type I is given in Fig. 4. In the upper two branches representing real and imaginary part of a zero mean complex Gaussian process two harmonic functions are added. These additive components stand for the line of sight part in the channel transfer function with ρ, f_ρ and φ_ρ being its amplitude, frequency and phase, respectively.

In order to not only describing the short term but also the long term behavior of the channel a third Gaussian process (lowest branch in Fig. 4) is required. This $\mathcal{N}(0, 1)$-distributed process is turned into a lognormal process by adding and multiplying the constants μ and σ, respectively, and by the exponential element behind it. The appropriate selection of μ and σ is described in detail in [WG98].

If one wants to consider path loss additionally which is necessary for the simulation of power control mechanisms the output of the Suzuki model has to be multiplied by a certain attenuation factor. In the literature many approaches can be found. In this work a prediction model according to Lee [Lee93] was employed which was derived from a measurement campaign conducted in the USA. Here, the attenuation factor h_{loss} is given by

$$h_{loss} = (\frac{r}{r_0})^{-\beta}(\frac{f}{f_0})^{-n}\kappa_0. \tag{9}$$

The parameters can be taken from Lee's work mentioned above. Interestingly, the exponent of the distance dependent attenuation β varies from $3,05$ to $4,31$ depending on the propagation environment. Notice that additional scattering effects raise β considerably compared to pure path loss ($\beta = 2$).

2 Determination of the instantaneous SNIR

In this section describes how the mathematical equations and models mentioned above can be used to build up a bit error generator. Therefore, all the parameters in eq. (4) have to be known. The transmit power P_S is normalized ($P_S = E\{\underline{SS^*}\} = 1$). Hence, all the noise powers in eq. (4) have to normalized, too. The amplitude amplification $A(t)$ can be chosen dynamically during the simulation run but it can also be fixed ($A(t) = 1$) if power control is neglected. The realization of the channel process was discussed extensively in section 1.3.

The only parameters left are the noise powers P_{N_i}, P_{N_e} and P_{N_0} which are normalized to P_S. P_{N_0} is an arbitrary parameter representing additive noise (AWGN) introduced by thermal and background noise. The internal interference P_{N_i} cannot be determined directly and depends on the implementation of the baseband processing in the receiver. As an exact simulation of this part of a mobile radio system is not the intention of this work the parameter P_{N_i} needs to be specified reasonably or extracted from a stream driven simulation.

This can be performed very easily according to eq. (7). If one knows the error floor BER_{floor} of a data transmission link (BER with $\gamma_{AWGN} \to \infty$) or if one specifies it the simple horizontal projection of the error floor on the AWGN performance curve corresponding to the actual modulation scheme delivers γ_{floor} directly. In case of BPSK or QPSK even an analytical relation applies

$$BER_{floor} = \frac{1}{2}\text{erfc}(\sqrt{\gamma_{floor}}). \tag{10}$$

One of the most complicated tasks when implementing eq. (4) is the dynamical calculation of the external interference P_{N_e}, because it depends on many factors that can change during the simulation run. Here, we present a solution enabling the simulation of CCI as well as MAI. Therefore, the entire network of all N participating nodes is considered to be fully meshed. Between two nodes i and j there exist independent and reciprocal mobile radio channel processes $H_{ij}(t)$, so that

$$\begin{aligned} E\{H_{ij}H_{kl}\} = 0 \quad & \text{für} \quad i \neq j \vee k \neq l \\ E\{H_{ij}H_{kl}\} = 1 \quad & \text{für} \quad i = j \wedge k = l \end{aligned} \tag{11}$$

and

$$H_{ij} = H_{ji}. \tag{12}$$

The additive superposition of all arriving wave fronts being transmitted by the N nodes in the network and being received at the ith node can be expressed conveniently in vector-matrix notation

$$\mathbf{r} = \mathbf{Hs}, \tag{13}$$

with $\mathbf{r} = (r_1(t), r_2(t), \ldots, r_N(t))^T$, $\mathbf{s} = (s_1(t), s_2(t), \ldots, s_N(t))^T$. Where $s_i(t)$ and $r_i(t)$ stand for the transmitted and received signal at the ith node, respectively. Thus, in case of normalized transmit power ($P_S = 1$) $s_i(t) = A_i(t)$ applies. The elements of the main diagonal H_{ii} have to be set to zero as a node does not transmit to itself.

The vector $\mathbf{r}$ can be employed to simulate measurements of the received power level at the stations. It can also be used to calculate the external interference at node i when receiving from node j

$$n_{e_{ij}}(t) = r_i(t) - h_{ij}(t)s_j(t). \tag{14}$$

The introduction of the cross-correlation factor c and the transmit matrix $\mathbf{S}$

$$\mathbf{S} = \begin{pmatrix} 0 & s_1 & s_1 & \cdots & s_1 \\ s_2 & 0 & s_2 & \cdots & s_2 \\ \cdots & s_3 & 0 & s_3 & \cdots \\ \vdots & & & \ddots & \vdots \\ s_N & s_N & \cdots & s_N & 0 \end{pmatrix} \tag{15}$$

yields the external interference matrix $\mathbf{N_e}$ with the elements $n_{e_{ij}}$

$$\mathbf{N_e} = c\mathbf{HS}, \tag{16}$$

where c denotes a cross-correlation factor representing the non-zero cross-correlation of the eventually employed spreading codes which can be set to 1 in systems not using spread spectrum techniques.

3 Modeling the physical layer

Now that all variables in eq. (4) are determined the instantaneous SNIR $\gamma(t)$ can be calculated. In case of BPSK the bit error rate can be derived immediately using eq. (10). Therefore, a random variable has to be realized which is uniformly distributed over the interval $[0, 1)$. By comparing this random number with the instantaneous BER one can decide whether a bit error has occurred or not.

When dealing with multilevel modulation schemes there is no analytical relation between bit error rate and SNR because this relation depends on the coding of the signal space (i.e. Gray code). Hence, a decision stage has to be implemented. Therefore, a symbol of the signal space is chosen according to a uniform distribution. Then a zero-mean complex Gaussian random number ($\mathcal{N}(0, \sigma^2)$) is added having a variance σ^2 that corresponds to the actual value of $\gamma(t)$. If the signal space is normalized ($P_S = 1$) the following equation applies

$$\sigma^2 = \gamma(t)^{-1}. \tag{17}$$

With the known signal space coding and the decision boundaries one can easily determine whether and how a symbol has been corrupted by the channel and which bit error pattern results.

Channel coding was embedded in a strongly simplified way but still having properties resembling the ones of block codes like RS- or BCH-codes. Using this kind of codes the error correctability measured in symbols per block can be specified directly as a design parameter. Hence, it is straight forward to calculate the average number of correctable bit errors in a data packet of a certain length. If the number of errors in an arriving data packet exceeds this threshold the data packet has to be discarded.

4 Simulation results

The presented mathematical model of the mobile radio channel was implemented in OMNeT++ and embedded in an existing simulation of the wireless LAN standard IEEE 802.11. A simulation model of the physical layer of same standard in the stream driven simulation tool COSSAP served as a basis for comparison.

It was investigated whether the first and second order statistics of the generated bit error rate process of both simulation systems match. Fig. 5 shows the average BER vs. SNR as an example for the first order statistics, where SNR denotes γ_{AWGN} from eq. (5). The simulations were conducted on an OFDM link with 4-PSK modulation using a rural channel according to the specification in COST 207 [COS89].

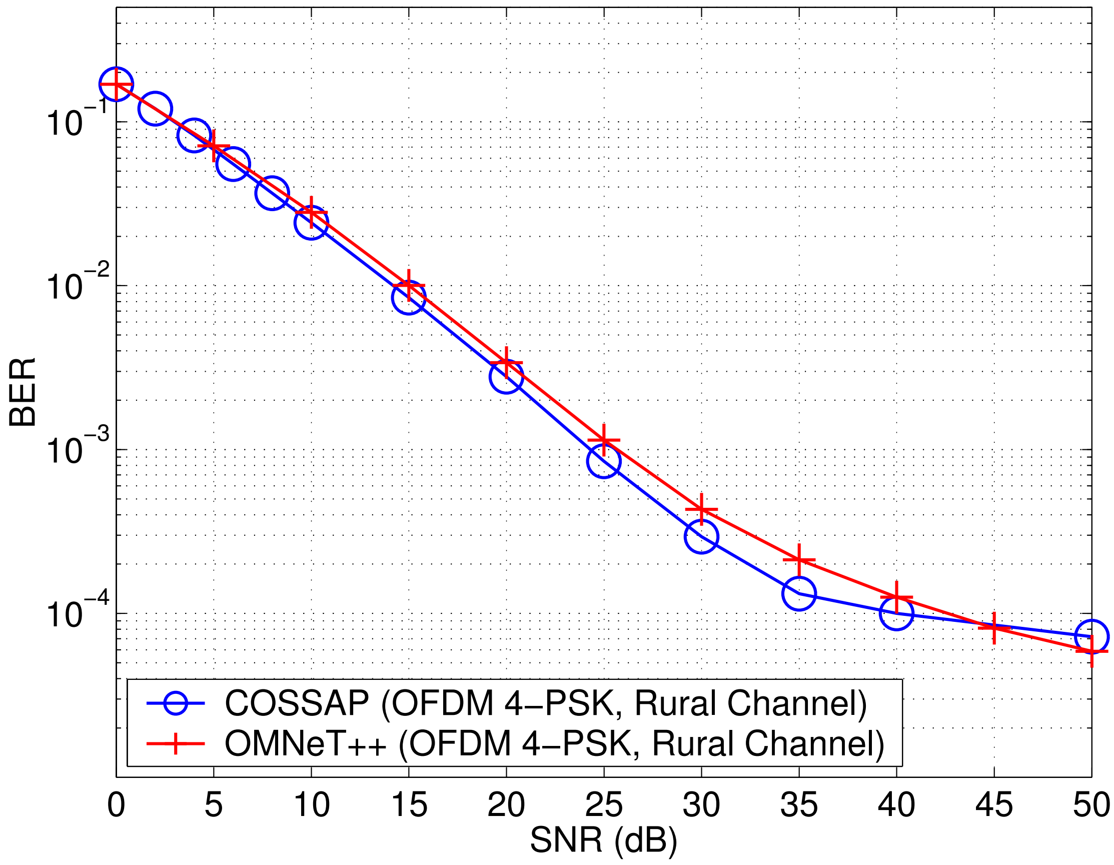


Fig. 5. Average BER as function of SNR

One can recognize a good match of both curves. The slight differences between them are due to the abstraction of the channel estimation that was implemented in detail under COSSAP while it was reduced to γ_{floor} from eq. (7) under OMNeT++. However, this mismatch can be neglected for the functional flow in higher protocol layers. Furthermore, the simulation duration using our channel model only amounted to one tenth of the duration of the stream driven model.

Fig. 6 depicts that the second order statistics of the presented model match the stream driven simulation curve as well as the theoretical curves utilizing an ideal Jakes model. Here, we considered the autocorrelation function of the bit error rate process in the interval 0 to 1ms. The solid and the dashed line represent the stream driven and the discrete event approach, respectively, while the dash-dotted line stands for the ideal analytical curve. The simulations were conducted on a BPSK-transmission over a flat fading Rayleigh channel. The maximum Doppler frequency was 100Hz and the Doppler spectrum was assumed to be Jakes shaped.

References

Cla68. R.H. Clarke. A statistical theory of mobile-radio reception. *Bell Syst. Tech. Journal*, 47, August 1968.

COS89. 207 COST. Digital land mobile radio communications. Technical report, Office for Official Publications of the European Communities, 1989.

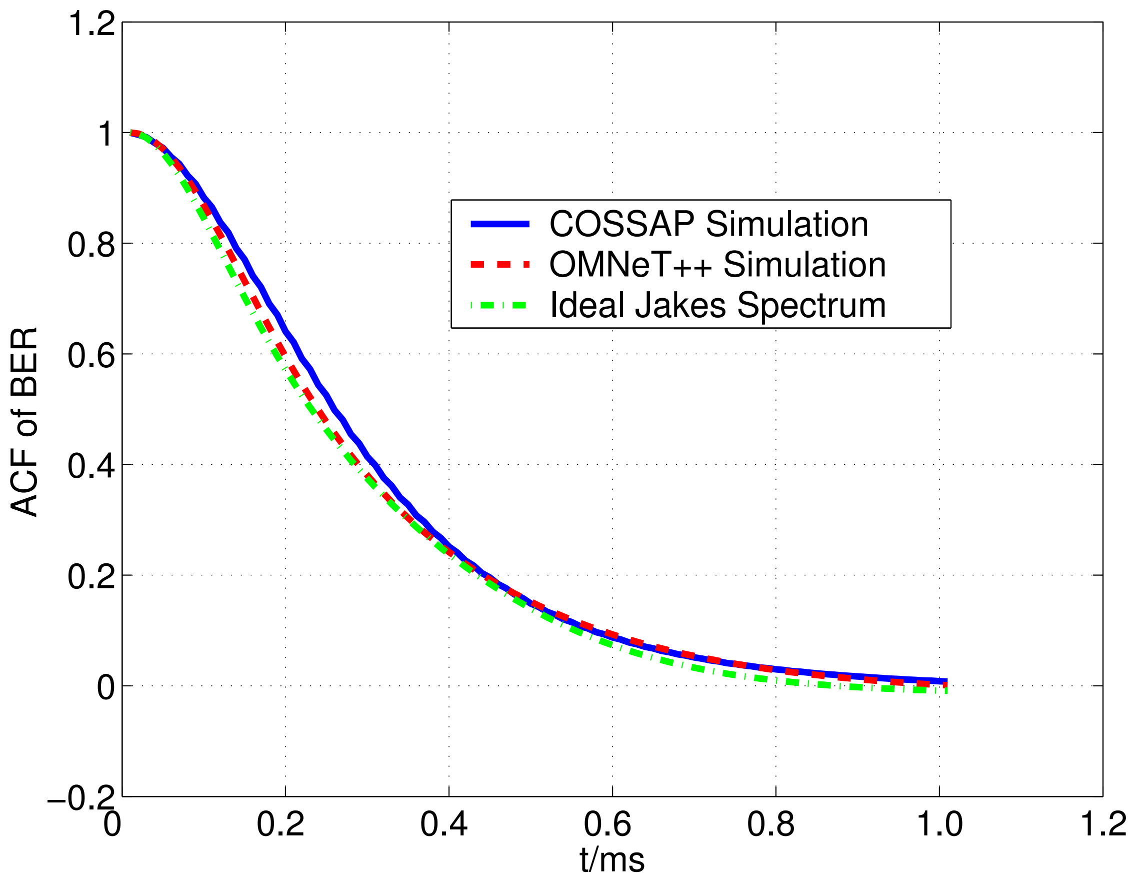


Fig. 6. Autocorrelation function of the BER process

Jak93. W.C. Jakes. Microwave mobile communications. *IEEE Press*, August 1993.

Lee93. W.C.Y. Lee. *Mobile Communications Design Fundamentals*. John Wiley and Sons, New York, 1993.

Pae94. M. Paetzold. A deterministic model of a shadowed rayleigh land mobile channel. *Proc. 5th IEEE Int. Symp. Personal, Indoor and Mobile Radio Comm., PIMRC'94, Den Haag, Netherlands*, September 1994.

PKL97. M. Paetzold, U. Killat, and F. Laue. Modeling, analysis and simulation of nonfrequency-selective mobile radio channel with asymmetrical power spectral density shapes. *IEEE Trans. on Veh. Technol.*, 46, May 1997.

PKL98. M. Paetzold, U. Killat, and F. Laue. An extended suzuki model for land mobile satellite channels and its statistical properties. *IEEE Trans. on Veh. Technol.*, May 1998.

Ric45. S.O. Rice. Mathematical analysis of random noise. *Bell Syst. Tech. Journal*, 24, January 1945.

WG98. W. Wiesbeck and N. Geng. *Planungsmethoden für die Mobilkommunikation*. Springer, 1998.

A HiperLAN/2 simulation model in OMNeT++

Daniel Hollos[1] and Holger Karl[1]

Telecommunication Networks Group
Technical University Berlin
Berlin, Germany
hollos@ee.tu-berlin.de, karl@ee.tu-berlin.de
http://www-tkn.ee.tu-berlin.de

Abstract. This paper describes the implementation of an OMNeT++-based HiperLAN/2 simulation model. The simulation models the HiperLAN/2 standard with a large degree of accuracy, in particular, in the DLC layer. It is designed to be extendable and flexible and incorporates HiperLAN/2 protocol extensions for multi-hop relaying. It is publicly available.

1 Introduction

Wireless communication has received increasing interest in the recent year. The wireless local area networks (WLANs) market is currently dominated by IEEE 802.11-based products [1]. Advantages of IEEE 802.11 networks are simple setup and reasonable performance in typical network conditions. A possible competitor to IEEE 802.11 is the HiperLAN/2 standard [2]. It promises higher data rates than the current IEEE 802.11 networks, reaching up to 54 MBit/s in optimal circumstances. A number of differences between IEEE 802.11 and H/2 lend credibility to this claim, in particular, the use of a centralized medium access scheduling as opposed to the distributed algorithms used in 802.11 (see Section 2 for more details). This control is performed by a so-called "central controller", which is responsible to orchestrate the communication between a number of terminals, together forming a HiperLAN/2 cell. In an infrastructure-based setup, the central controller would usually be the base station (as it is usually connected to a fixed power supply), in an ad-hoc network, any terminal can be elected to perform these control functions.

The use of such a centrally controlled medium access opens possibilities for additional protocol extensions. One such extension is the introduction of relaying capabilities into HiperLAN/2, using intermediate terminals to communicate between the central controller and another terminal, allowing to reduce the radiated power used by all terminals. The goal of such relaying protocols would be to improve the total capacity of a wireless network by reducing interference [3, 4] as well as to increase the energy efficiency of the communication (which is important as mobile devices usually only have a limited power supply). We are pursuing both these goals in the context of the IBMS [2] project [5].

In order to assess such claims, as well as to experiment with such extensions to and modifications of the H/2 standard, a simulation model of the standard is needed that allows to efficiently evaluate the performance of a system. Such a model must be extensible, modular, easy to understand, and fast. In the context of the IBMS [2] research project we have developed such a simulation model. The essential characteristics of this model are described in this paper; for more information (and for the source code) please refer to the simulator homepage [6].

The following section gives a brief overview of H/2. In Section 3, the simulation model itself is described. Section 4 summarizes our contribution and discusses possibilities for future work.

2 The HiperLAN/2 System

The HiperLAN/2 (H/2) WLAN system standardized for the 5.2 GHz range with modulation types of up to 54 Mbit/s. The goal of this standard is to maximize the utilization of the radio channel. The main mechanism — and the largest difference to IEEE 802.11 — is the use of a a priori scheduled medium access: Among a set of stations (a so-called cell), one station is declared to be the central controller (CC). Any station that wants to communicate with another station has to announce this to the CC, which will grant time slots in a periodically repeated frame to this station. Hence, H/2 uses a connection-oriented, centrally scheduled TDMA to organize the medium access. Main benefits are collision-free data traffic (100% efficiency, no hidden or exposed terminal problems) and simple support for priorities or QoS requirements.

Scheduling happens on the basis of a frame, which is divided into different phases (compare Figure 1), which are again divided into cells of fixed length. In the first phase, the central controller broadcasts administrative information (BCH and FCH), in particular, which terminal is allowed to transmit to whom at what time and for how long. As this information is available to all terminals, perfect channel access can be organized. While this idea is very simple, it is the source of performance and flexibility of H/2.

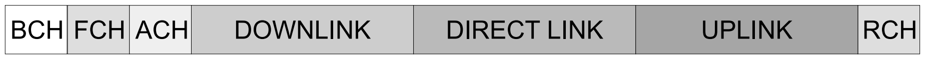


Fig. 1. H/2 MAC frame structure

One example of this flexibility is the so-called direct-link traffic: As it is possible for the CC to assign a time slot to one terminal as a sender, to another terminal as a receiver, terminals can communicate directly with each other without having to send the data via the CC. Usually, the sequence of transmissions is organized such that after the frame's initial administration phase, downlink traffic is scheduled, then direct-link traffic, and uplink traffic; there is also a random access time-interval at the end of the MAC frame for associations and other infrequent, unscheduled requests (resulting in five phases in total).

Using this basic mechanism, a number of additional capabilities are included in the H/2 standard (e.g., automatic frequency selection, multicast, automatic CC selection, etc.). Also, the CC can request channel measurements from any terminal, describing the channel characteristics between any two terminals (which is important for multi-hop extensions of H/2, see Section 4, and for the parallel transmission extension of the CC scheduler, see Section 3.2). Other possible add-ons are sleep modes for terminals or the interconnection of separate H/2 cells [7].

3 Features and Possibilities

Main goals of the simulator design were flexibility and extensibility as well as a correct capturing of reality: No "incorrect" information transfer or any other methods which are possible in a simulation environment, but not possible according to real life or the standard, are used. The following Subsection 3.1 describes the simulation of standard H/2 functionality, Subsection 3.2 exemplifies the simulator's extensibility by outlining some functions we have introduced in the course of our multi-hop research.

3.1 Simulator Structure

This simulator was made for research purposes; we payed attention to separate different parts, and tried to keep it open to future developments. Therefore, all major functionalities have their own separate modules; these modules communicate with each other using OMNeT++ messages.

The main configuration file (omnetpp.ini) defines the number of cells, number of terminals, the data source description file and the initial location file.

The initial location file contains the identifiers (MAC_IDs) and initial positions of all terminals. Note that some of them can be turned off and later on, so this number should be the maximum number of the terminals. The positions are "initial" as it is possible that a terminal moves during a simulation; however, mobility is currently not yet supported.

In the following subsections we give a brief description of all modules included in the simulator, its structure is outlined in Figure 2. As the primary focus of our own research is on the link and routing layer, these functions are modeled with the largest degree of detail; lower layers are comparably simple, but it is easy to replace these layers by more sophisticated modules.

The lowest layer in our simulator is an abstraction of the radio channel. The basic idea is to represent the physical layer along with its error behavior in a single module. All terminals are grouped into cells (compound modules), which are in turn connected to the channel module. An example of such a layout can be seen in Figure 3; Figure 4 shows how a radio cell consists of a central controller and a number of terminals.

A terminal is turn again structured as a compound module, consisting of a load generation module (also used as a data sink), the network stack as such and a control module (Figure 5). The network stack is the last compound module used in the simulator, it contains simple modules for the datalink, network and transport protocol layers (Figure 6).

The Configuration Distributor This module is always called first when starting a simulation, and later any time when the configuration changes. For example, a terminal can

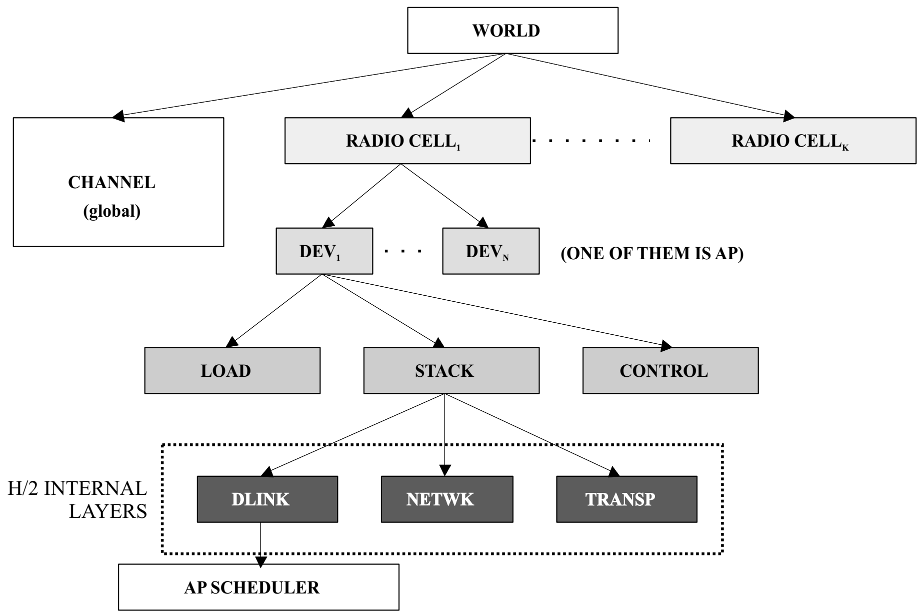


Fig. 2. Simulator structure

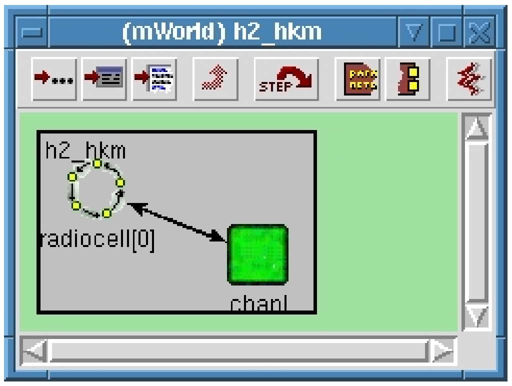


Fig. 3. Screenshot of the physical channel and a single radio cell

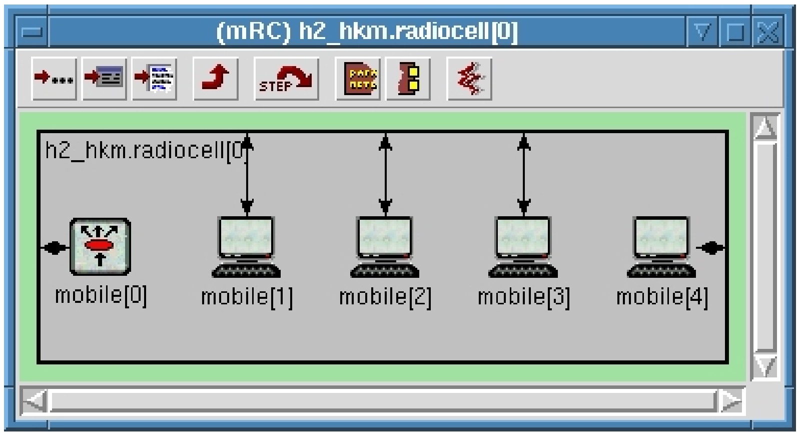


Fig. 4. Screenshot of a radio cell compound module, containing a single controller (mobile[0]) and four terminals

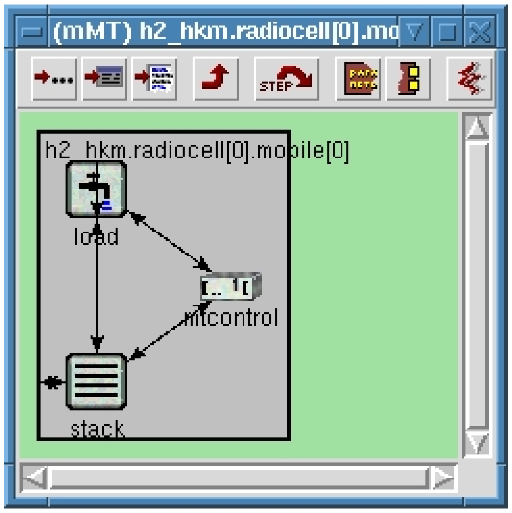


Fig. 5. Screenshot of a terminal compound module

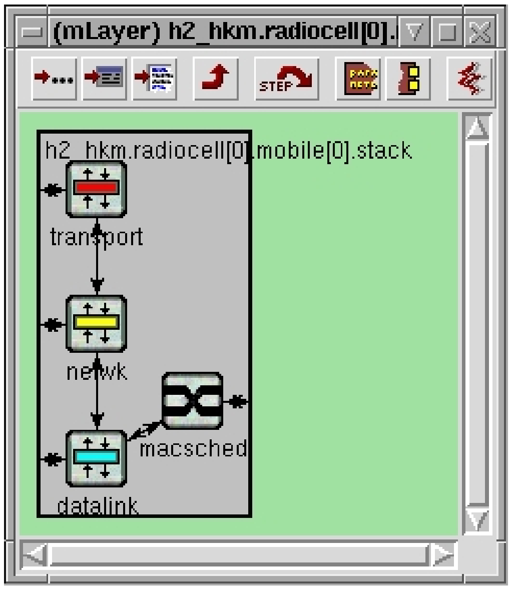

(mLayer) h2_hkm.radiocell[0].
STEP
h2_hkm.radiocell[0].mobile[0].stack
transport
netwk
macsched
datalink

Fig. 6. Screenshot of a stack compound module, showing simple modules for the individual network layers.

be turned into a central controller by calling a function in this module. It is connected to all layers and distributes the new state to them.

The Physical Medium Simulation Module The terminals send their data to the Physical Medium Simulator Module called "channel" (common for the entire scenario). This module is responsible for determining whether or not a transmission succeeds or a packet is lost. To do so, this module collects packet transmissions and calculates the delay to all other terminals. For each destination terminal, the channel gain and the interference level is then calculated. Based on this signal to interference/noise ratio, a packet error rate (PER) can be calculated [2]. The PER is used to decide (with a simple Bernoulli random variable) whether the packet is correctly received by the terminal; if not, an error indication is sent to the terminal (akin to a failed CRC check). In case the received power is below the receiver sensitivity, the packet will not be delivered at all.

Each packet will also increase the interference of the non-destination terminals as determined by the respective channel gain. As a simplification, when calculating the PER (Packet Error Rate) we use the highest interference level determined during that particular packet.

The data collection, parallel interference calculation and all other physical-layer related calculations are put to a subclass of the channel module. Therefore, applying a more sophisticated physical layer model is a matter of replacing the channel subclass of this module.

The Physical Layer This is a very simple layer, one instance per H/2 device, which adds the basic H/2 preambles to the MAC packets.

The channel module always needs some parameters which are given in real life, but not in the simulation: for example, the transmission power, the modulation type to use, terminal ID (for simulation administrative reasons), etc. These parameters are also set by this layer; they have a separate, unique structure in each message type.

The DLC Layer The data link control (DLC) layer contains the basic data sending and reception functionalities, like handling data initiated by the load generator, constructing resource requests, decoding the administrative information received at the beginning of a frame (BCH and FCH cells), sending the data at the scheduled time, etc.

The DLC layer (outlined in Figure 7) basically contains two major subclasses: APControl and MTControl. They are for the different AP and MT functionality, but as a subclass they contain MTInstance class(es). An MTControl has only one MT subclass, the APControl has one for each associated MTs and one for itself. Changing a common MT functionality is a matter of changing a function in the MTInstance class. The MTInstance classes contain several DLCC queues: send, received, sending_in_this_frame, received_in_this_frame. When an MTInstance receives data from the FCH which is valid for it, allocates space in one of its _in_this_frame DLCC queue immediately. This structure reflects reality, since MTs have time after the FCH to do administrative procedures, but not later when receiving by 54Mbit/s rate. The DLC layer also has the CC scheduler as a subclass, which is activated only when the node is indeed a CC (see Section 3.2).

The CC scheduler proceeds in two steps, implemented in two separate classes: one for handling priorities, QoS demands, parallel transmission support, etc., computing a schedule describing when each terminal is allowed to transmit (`Compute_Schedule)`. The second module constructs the MAC frame based on this schedule, keeping the H/2 rules defined in the standard (`Build_FCH`). Thanks to this structure, new scheduling algorithms (priorization, parallel transmission, or any other optimization method) can be applied without having to worry about complex H/2 MAC frame structure rules.

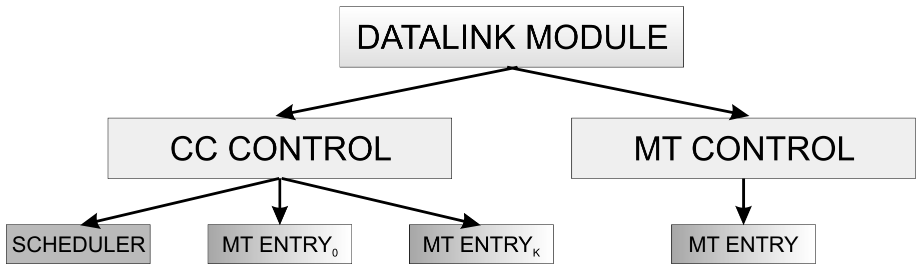


Fig. 7. DLC layer structure

All nodes have the same DLC, RLC, etc. modules.

The Network Layer This is an internal network layer, part of the H/2 RLC; it has the database with DLCC connections for that terminal. When a data packet comes from the load generator, this layer translates source- and destination MAC_IDs to streams with DLCC_IDs; when data has been received by the DLC layer, it translates DLCC_IDs to source- and destination MAC_IDs.

Generating Load Currently, two simple load generation methods are implemented: Constant bitrate (CBR), and Poisson process (PPS). The load generation parameters are contained in a file (which is specified in omnetpp.ini). In CBR mode the start- and end time, the bitrate of the data flow, and the DLCC_ID can be specified. In PPS the starting- and ending time, and the parameter λ of the arrival process can be specified. The system generates as many load modules per terminals as were described in the data sources description file. Other load generators can be easily integrated.

3.2 Extra Features

Here we describe the features we added to the simulation and are not present in the standard.

The Parallel Transmission CC Scheduler Since the H/2 provides direct, scheduled data transmissions between terminals in a cell, and the CC has all information to decide whether a parallel transmission is possible or not (based on channel measurements provided by the terminals, see Section 2), we developed a scheduler class (extending the one described in Section 3.1) which calculates possible parallel transmission pairs based on a graph coloring algorithm. Using this class the `Build_FCH` class is able to schedule multiple transmissions at the same time.

The Single Relay Protocol When a terminal is located on the edge of the cell, it has to use its highest transmission power to reach the CC. The single relay protocol enables the system to use one relay station for a transmission between the CC and a "far" terminal. The protocol fits into the H/2 standard and allows dynamic route path change even in every MAC frame without additional protocol overhead. In this case, a terminal located far away will use less transmission power (exploiting non-linear channel characteristics); it also dramatically reduces the inter-cell interference. The protocol realization itself is included in the CC scheduler `Build_FCH` class; the path calculation has to be done in the class `Compute_Schedule`. This work is part of our ongoing research in the context of the IBMS^2 project [5].

4 Conclusions

This paper has described an H/2 simulator, capturing essential parts of H/2's medium access and link layer functionalities. Its design lends itself to simple modifications and to introduction of new functionalities, which has been demonstrated by introducing a single-relay protocol and non-standard cell schedulers. We believe that this simulator could be useful to a broader community of researchers.

Currently, we are developing an IP convergence layer on top of the actual H/2 standard in order to be able to use realistic traffic models based on IP traffic for performance evaluation. This convergence layer will be integrated with the TCP/IP suite available for OMNeT++ [8]. In the long run, we are interested in incorporating true ad-hoc functionality such as dynamic CC selection, terminal handover between different cells and inter-cell connection support (H/2 multihop extension). Also, suitable mobility models should be integrated in the simulator. It will be a challenging task to take into account mobility-induced handovers between different CCs, as these are represented by compound modules. It would imply to remove a compound module representing a node from a cell compound module and reinserting it into another cell module in order to maintain the general structure of the simulation setup. In addition, this process has to closely coordinated with the protocol-specific processing of handover. It would be an interesting question to see if such functionalities are generally useful to the modeling of mobile ad-hoc networks; examples for such a generalization would be cluster-based routing protocols for ad-hoc networks, which also superimpose a cell-like structure upon an ad-hoc network. A convenient handling of such problems would be useful to OMNeT++-based simulations of ad-hoc networks.

More specifically to the OMNeT++ tool, a particular problem that we encountered was the simulation of the physical layer. Taking into account interference generated by other terminals as well as noise to compute packet error rates proved to be feasible only by delegating these tasks to a separate module which is responsible for the entire simulation setup; using OMNeT++ channel abstraction did not turn out to be useful. In general, the development of more sophisticated and customizable channel abstraction would prove rather useful in many respects. Besides the notion of simulation-wide (or cell-wide) computation of interference discussed here, also bit error models might prove to be useful, e.g., models which can express autocorrelation within the bit error process [9].

References

1. : Ieee 802.11 (iso/iec 8802-11:1999). IEEE Standards for Information Technology (1999) Available via `http://standards.ieee.org/getieee802/`.
2. Khun-Jush, J., Malmgren, G., Schramm, P., Torsner, J.: HIPERLAN type 2 for broadband wireless communication. Ericsson Review **2** (2000) 108–119 `http://www.ericsson.com/review/2000_02/files/2000026.pdf`.
3. Karl, H., Mengesha, S.: Analyzing capacity improvements in wireless networks by relaying. In: Proc. IEEE Intl. Conf. Wireless LANs and Home Networks, Singapore (2001) 339–348
4. Mengesha, S., Karl, H., Wolisz, A.: Improving goodput by relaying in transmission-power-limited wireless systems. In: Proc. of Informatik 2001-31. Jahrestagung der GI, OCG, Austria, Vienna (2001)
5. : Webpage of the IBMS^2 project. (http://www.ibms-2.de)
6. Hollos, D., Karl, H., Kubisch, M., Mengesha, S.: Hiperlan/2 simulator homepage. `http://www-tkn.ee.tu-berlin.de/research/H2Simulator/` (2001)
7. Peetz, J.: HiperLAN/2 multihop ad hoc communication by multiple-frequency forwarding. In: Proc. of Vehicular Technology Conference (VTC) Spring 2001, Rhodes, Greece (2001)

8. : Omnet++ internet protocol suite. (Webpage at `http://cvs-int.etec.uni-karlsruhe.de/omnetpp/model-doc/ip-suite.html`)
9. Willig, A., Kubisch, M., Wolisz, A.: Measurements and stochastic modeling of a wireless link in an industrial environment. Technical Report TKN-01-001, Telecommunication Networks Group, Technische Universität Berlin (2001)

Session 4:
Future OMNeT++

Java: Future Tools Platform for OMNeT++?

András Varga
Department of Telecommunications
Budapest University of Technology and Economics
andras@whale.hit.bme.hu

Abstract

Graphical user interfaces in OMNeT++ [1] are currently based on Tcl/Tk. Tcl/Tk does not seem to be a good long term choice: while it has merits in rapid UI prototyping, it is not really suitable for large scale projects, and it has seriously fallen behind contemporary GUI development trends and other GUI toolkits. Further alarming signs are the low level of development activity in the Tcl/Tk CVS and the huge gap between their roadmaps and what has actually been implemented from them.

This paper proposes that Tcl/Tk based parts of OMNeT++ be replaced by Java/Swing components. The paper proposes that the NetBeans open-source IDE framework be used as a base for a future OMNeT++ IDE. The OMNeT++ IDE may contain a graphical NED editor (GNED), a graphical FSM editor, a project manager for compiling and linking simulations and possibly other components. The primary advantage from using NetBeans is that a large amount of work can be spared by not having to spend efforts on developing our own user interface framework. The paper also proposes that Tkenv be replaced by a JavaEnv on the long term. The first step towards building JavaEnv is creating a JNI-based Java wrapper around the OMNeT++ simulation kernel classes.

These plans cannot be realized without collaborative effort from the community. A possible way to go is to implement parts of the project as student projects at universities.

Java and OMNeT++

Why Java?

In current OMNeT++ releases, all GUI components are built using Tcl/Tk. While Tcl/Tk proved to be very effective for GUI development, it is not suitable as a long-term choice. First, Tcl doesn't scale well for large-scale development. Variables are not typed, there is no support for object-oriented concepts and because it is an interpreted language, there are no compile-time checks on the code. Second, Tk has seriously fallen behind contemporary GUI development kits (Qt, Gtk or MFC), and there is no sign of its catching up. It has but the most basic widgets: even combo boxes, spinboxes or tabbed dialogs have to be assembled from other GUI elements, not to speak of dockable toolbars or tables. An alarming sign is that while open-source development flourishes these days and there are several very large-scale open-source projects (KDE, Gnome, JBoss, the Apache and Jakarta projects, etc), there is very little activity on Tcl/Tk.

As a replacement for Tcl/Tk, we might use Qt, Gtk or MFC and C++. However, Java and the Swing GUI toolkit appear to be a better choice for several reasons.

- Swing offers all contemporary GUI elements.
- Java/Swing is completely platform-independent. The requirement of platform independence eliminates MFC from possible choices, and also Qt and Gtk are not really strong choices.

Also, Java has significant advantages over C++:

- much faster development cycle because of faster compilation
- much shorter debugging time (no segfaults, usually exception stack trace immediately gives enough info to locate bug)
- huge amount of libraries and APIs ready for immediate use (Regexp and XML, to mention two practically important ones)

- huge amont of reusable components (GUI and other)

Java/Swing can be a replacement only for the components currently based on Tcl/Tk. Existing Tk-based components need to be supported until Java-based ones become mature.

OMNeT++ Integrated Development Environment

Motivation

Instead of creating a standalone GNED and possibly other GUI-based OMNeT++ tools as standalone programs, it is better to build a single integrated development environment where these components are plugins. The OMNeT++ IDE may contain a graphical NED editor (GNED), a graphical FSM editor, a project manager for compiling and linking simulations and possibly other components.

NetBeans from Sun

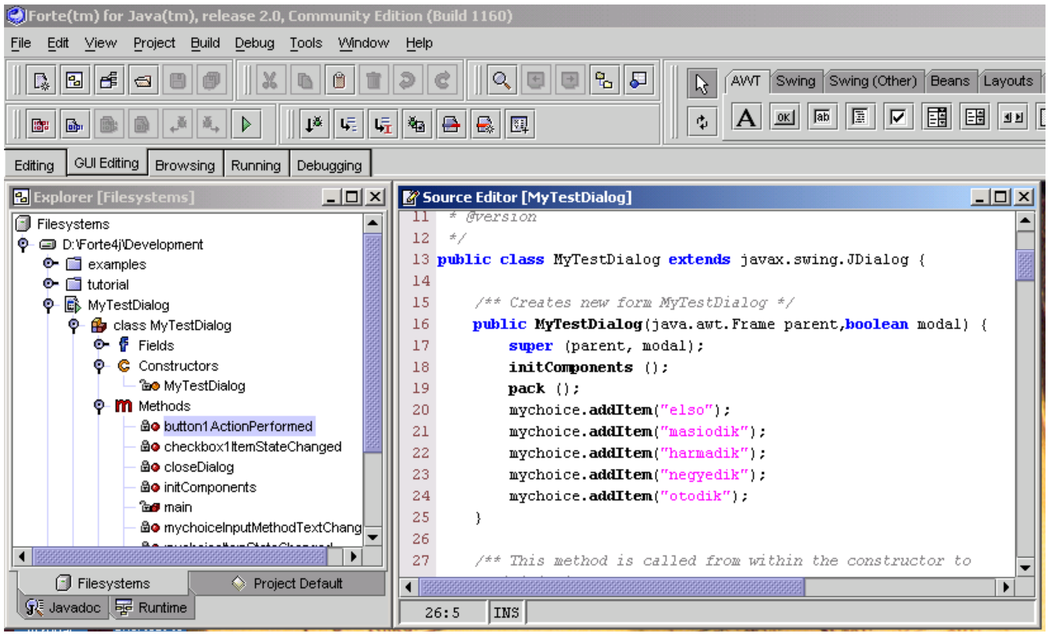
NetBeans [2][3] from Sun Microsystems is a portable, Java-based IDE framework. NetBeans is extensible, modular and standards-based, and what is equally important, it is open source. Because of these reason, NetBeans is a possible candidate to be used as a base for an OMNeT++ integrated development environment. NetBeans is available from www.netbeans.org.

Because NetBeans is completely modular, developers can:

- Add modules that provide editing, debugging, syntax coloring, error highlighting, etc. The editor can work with C, C++, UML, IDL, XML, Java and other languages.
- Switch any IDE modules on/off. By switching off unneeded modules, the IDE consumes less memory and no longer offers unnecessary information and actions.
- Write modules that add new features or replace functionality in the IDE.
- Update the IDE online through the Update Center

A recent addition, the standards-based Metadata Repository makes it easier to build modules to support other programming languages.

The NetBeans IDE has all GUI elements and features one expects from a contemporary IDE (explorer tree, syntax-coloring source editor, component toolbars, etc.). The following figure is a screenshot of the NetBeans IDE.


Forte(tm) for Java(tm), release 2.0, Community Edition (Build 1160)
File Edit View Project Build Debug Tools Window Help
AWT Swing Swing (Other) Beans Layouts
Editing GUI Editing Browsing Running Debugging
Explorer [Filesystems]
Filesystems
D:\Forte4j\Development
examples
tutorial
MyTestDialog
class MyTestDialog
Fields
Constructors
MyTestDialog
Methods
button1ActionPerformed
checkbox1ItemStateChanged
closeDialog
initComponents
main
mychoiceInputMethodTextChang
Filesystems
Project Default
Javadoc
Runtime
Source Editor [MyTestDialog]
26:5
INS

```
 * @version
 */
public class MyTestDialog extends javax.swing.JDialog {

    /** Creates new form MyTestDialog */
    public MyTestDialog(java.awt.Frame parent,boolean modal) {
        super (parent, modal);
        initComponents ();
        pack ();
        mychoice.addItem("elso");
        mychoice.addItem("masiodik");
        mychoice.addItem("harmadik");
        mychoice.addItem("negyedik");
        mychoice.addItem("otodik");
    }

    /** This method is called from within the constructor to
```

NetBeans is extensible via plug-in modules written to the NetBeans OpenAPIs (15 API sets, over 400 classes). Virtually every aspect of the IDE is extensible. New functionality can be added or removed simply by adding or removing modules. The Core APIs are the following: Modules, Nodes, DataSystems, FileSystems, Explorer, Actions, Options, Execution, Compiler. The Metadata repository supports the MOF, XMI and OCL standards.

The proof that the extensibility concept really works is the large number and variety of projects (both commercial and open-source) that build upon NetBeans. While most projects are some sort of Java development environment (i.e. GUI builders, EJB or servlet/JSP editors), there are also UML editors and radically different ones like a commercial mine planning application or a data analysis environment for biologists. A sample of the over 30 projects listed on www.netbeans.org:

Forte for Java (Sun) • BEA Campaign Manager for WebLogicTM (BEA) • CocoBase an Object/Relational mapping tool (Thought, Inc.) • UML modeling tool (Gentleware AG) • Describe, an integrated UML modeling & Java development environment (Embarcadero Techn.) • DataMirror, bi-directional data transformation between XML, database and text formats (DataMirror Corp.) • iWarf Service Creation Environment (SCE), for creating telecom applications (Incomit) • an open source GIS application (Leeds University) • BioBeans, an integrated data analysis environment for biologists (University of Glasgow) • MINEX V integrated mine planning application (ECSI).

A NetBeans-based OMNeT++ IDE

How could NetBeans be used in OMNeT++? Some ideas:

- A GNED module could provide NED graphical and source editing.
- The same IDE could be used for developing simple modules. The corresponding C++ source editor module has probably already been created within the NetBeans project.
- A graphical FSM editor could provide a front-end for writing state machine-based simple modules
- A “project manager” could control compiling and linking simulations, providing the IDE equivalent of opp_makemake and make. A ‘makefile’ module already exists within the NetBeans project.

The promary advantage of using NetBeans is that it would enable the developers to **concentrate on the core tasks** (e.g. creating the NED editor), instead of wasting energy on building their own GUIs.

GNED module for NetBeans

A GNED module for NetBeans could be developed in the following steps.

Preparation:

1. get familiar with NetBeans APIs: write an experimental NED editor (source editor only, with syntax highlighting)
2. explore the Metadata Repository, create representation of NED inside NetBeans (converge NED-XML data tree and MDR?)

Next steps (may go parallel):

3. create NED-XML "converters": NED parser, NED generator, XML parser, XML converter
4. build graphical editor on top of NED data classes

As for developers: The first two tasks are small enough so that they can be conveniently given out to students as one-semester assignments. The third task is also very small (the C++ implementation of the same components could be easily converted to Java). However, the fourth task (the actual NED editing) is of larger-scale, it would take several semesters if done as student projects.

Graphical FSM Editor

A graphical FSM editor module could serve as a front-end for writing state machine-based simple modules for OMNeT++. The graphical editor might feature a GUI like that of OPNET or NetSim++, and generate C++ code that uses the FSM macros.

The task is currently being advertised as student project by Ahmet Sekercioglu at Monash University, Australia.

Other NetBeans modules

- project manager (~makemake)
- analysis tools (Plove + more)

JavaEnv: Java-based graphical runtime environment

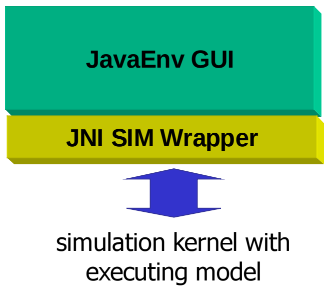
On the long term, Tkenv should be replaced by a Java-based GUI. This will enable a more powerful user interface. Also, the Java language will enable a much cleaner program design, thereby lowering maintenance costs.

JavaEnv might also be implemented as a plugin to NetBeans.

Java wrapper around SIM

JavaEnv will have to contain a layer that interfaces Java GUI code with the C++ simulation kernel. Part of this layer is a Java wrapper around OMNeT++ API classes. Every C++ class in the simulation kernel should have a Java wrapper – a Java class that has similar methods that the C++ class, and Java methods are implemented via the Java Native Interface (JNI) and call methods of the C++ class.

JavaEnv would be built using the Java wrapper classes, as illustrated by the following figure.



The Java wrapper can also be useful outside JavaEnv:

- It would enable embedding OMNeT++ simulations in Java apps.
- The same library might also enable writing simple modules in Java, should the need ever emerge.

Code for the Java wrapper should be automatically generated from the SIM C++ headers. A handy tool for this purpose might be the J2C++ program from IBM Alphaworks [4]; the resulting code could then be hand-tuned.

The task is currently advertised (being worked on?) as student project at University Karlsruhe.

Conclusion

NetBeans as OMNeT++ IDE

- GNED module (NetBeans infrastructure, NED-XML data structure)
- others: FSM Editor, etc.

JavaEnv

- 1st step: Java wrapper around simulation classes
- Java wrapper may have other uses (e.g. Java-based simple modules)

This project cannot be carried out without collaborative effort from the OMNeT++ community.

References

[1] OMNeT++ Discrete Event Simulation System. http://www.hit.bme.hu/phd/vargaa/omnetpp.htm

[2] NetBeans. http://www.netbeans.org

[3] The NetBeans Tools Platform, white paper.
http://www.netbeans.org/download/NetBeansToolsPlatform.pdf

[4] IBM AlphaWorks. http:// www.alphaworks.ibm.com

Second Generation NED

András Varga
Department of Telecommunications
Budapest University of Technology and Economics
andras@whale.hit.bme.hu

Abstract

This paper introduces NED-2, the second generation of OMNeT++'s NED language. In addition to enhancing the current NED, NED-2 provides important new features such as possibility to define messages (also known as "message subclassing"). NED-2 will be implemented with modular compiler architecture, centered around XML. A specification draft and some source code are already present.

NED-2 Goals

NED-2 [2] will be an improvement over the current NED used in OMNeT++ [1] in the following areas. It will support defining message types with their contents; it will enhance the parameters, gates, channels and connections. It will provide inheritance for components (simple and compound modules, channels, networks and messages). It will support enums. NED-2 will enable automatic doumentation generation from the source code (similar to Javadoc). Expressions will be enhanced in several ways. Last but not least, it will have an XML [6] interface for easy interoperability with other systems.

This paper is not a substitute for the NED-2 specification draft. The latter contains several additional points that had to be omitted from this paper due to size limitations. Interested readers are strongly encouraged to read the draft specification and provide feedback.

Messages in NED

Overview

In OMNeT++, modules communicate by exchanging messages. Most simulation models need that messages have attached parameters or fields. Current OMNeT++ simulation models use cMessages with dynamically added cPar objects. This usually results in poor performance because it incurs the creation and deletion of too many objects. An additional disadvantage is that the code is difficult to read and maintain, because the content of messages is not obvious (parameter list is implicit in the code). The latter problem could be solved by defining message types separately – that is, by introducting typed messages.

A possible workaround is avoiding cPars and subclassing cMessage and adding new fields as C++ data members instead. This apparently solves both problems mentioned above, but has two drawbacks: one is that too much C++ code has to be written manually (there are several methods that must be overwritten when subclassing cMessage); the second is that data fields added in C++ are not visible for Tkenv. The second problem could be solved with writing custom inspectors for Tkenv, but that's also inconvenient.

The solution introduced in NED-2 is to describe messages with their fields as part of NED, then let the NED compiler (nedc) generate corresponding C++ classes from them. The idea is somewhat similar to CORBA IDL [5] and IDL compilers. The solution cures all the forementioned problems: it provides typed messages and low runtime overhead, the simulation programmer is freed from having to write a lot of C++ code (because nedc generates it), and also solves the Tkenv problem because nedc can also generate meta-info that Tkenv can use. It is also an improvement conceptually: now that messages are part of NED, NED code provides more complete information about module interfaces.

Example

An example NED-2 message:

```
message DynaPacket
{
    fields:
        short packetType;
        int srcAddress;
        int destAddress;
};
```

Technically, from a NED source file called foo.ned, the nedc compiler will generate foo_n.h and foo_n.cc. Simple modules will need to include foo_n.h and link with foo_n.cc. Given the above NED-2 fragment, the generated foo_n.h will contain code like:

```
class DynaPacket : public cMessage {
  protected:
    int srcAddress;
    int destAddress;
  public:
    DynaPacket(const char *name=NULL);
    DynaPacket(const DynaPacket& other);
    virtual ~DynaPacket();
    virtual const char *className() const {return "DynaPacket";}
    DynaPacket& operator=(const DynaPacket& other);
    virtual cObject *dup() const {return new DynaPacket(*this);}

    // field getter/setter methods
    virtual int getSrcAddress() const;
    virtual void setSrcAddress(int srcAddress);
    virtual int getDestAddress() const;
    virtual void setDestAddress(int destAddress);
};
```

For every field, nedc generates protected data members and getter/setter methods. The DynaPacket class can be used from simple modules as any other class:

```
#include "foo_n.h" // include generated header

void Client::activity() {
  ...
  conn_req = new DynaPacket("DYNA_CONN_REQ");
  conn_req->setPacketType(DYNA_CONN_REQ);
  conn_req->setSrcAddress(own_addr);
  conn_req->setDestAddress(server_addr);
  ...
  ev << "waiting for DYNA_CONN_ACK\n";
  conn_ack = (DynaPacket *) receive( timeout );
  ...
```

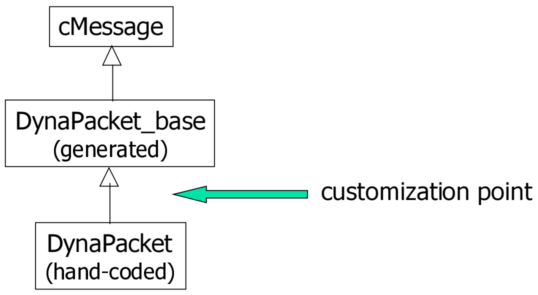
Inheritance

NED-2 supports inheritance for messages. In a subclassed message, one may add new data members or change the default values for fields. Single inheritance is supported; the generated C++ class hierarchy parallels the NED hieararchy. The NED syntax is similar to that of Java:

```
message DynaDataPacket extends DynaPacket { …
```

Customization

Because generated message classes may not always entirely fit the needs of the simulation programmer, NED-2 provides a way to customize the class via the Generation Gap object-oriented design pattern [3][4]. The idea is to generate an intermediate C++ base class which the user can customize by overriding its virtual member functions. This is illustrated by the following UML class diagram showing the C++ class hierarchy:



The need for using the Generation Gap pattern is signalled to the nedc compiler by the following syntax:

```
message DynaPacket
{
    properties:
        customize = true;
    fields:
        short packetType;
        ...
```

Additional features and experimental implementation

Currently there is an experimental perl-based nedc prototype that implements NED-2 messages. It provides support for the ideas described above, including:

- default base class can be cMessage (**`message`** keyword) or cObject (**`class`** keyword)
- single inheritance
- customization of generated class via the "generation gap" pattern
- support for generating cObject classes, non-cObject classes, structs
- enum support (enum name visible in Tkenv)
- extending enums (adding new elems to existing enum)
- nearly all C++ primitive types, one-dimensional arrays (both fixed-size and dynamically allocated)
- possibility to reference classes declared in external C++ code
- pointers (to both owned and not owned objects)
- "virtual" members (no data member generated, only pure virtual getter/setter methods)The prototype is fully functional (compromise affects error handling). Experiences (and with conversion, much of its code) can be directly reused in implementing the production nedc compiler. The prototype is part of the current OMNeT++ code snapshots in the Karlsruhe CVS.

Improvement on Existing Components

Inheritance

NED-2 supports inheritance for all components described in NED: simple and compound modules, channels, networks and messages.

Inheritance may add new parameters (fields), or set parameters to fixed values, thereby hiding them from derived and embedding components.

Inheritance of compound modules may add new gates, submodules and connections.

Gates, parameters, channels, connections

In NED-2, it is possible to create parameter vectors. Size of the parameter vectors must be given at the place of the parameter declaration (no “paramsizes” keyword). Parameter vector sizes can be defined with constants or expressions.

Parameters can be given a value at the place of their declarations. These fixed values cannot be overridden in enclosing components.

It is possible to declare gate vectors with their sizes defined at the place of the gate declaration (without using the “gatesizes”' keyword).

It will support specifying acceptable message types for gates (BONeS-like feature).

Channels can have arbitrary parameters in addition to the currently supported delay, error, datarate parameters. Channel parameters can be given a value at the place of their declarations. Inheritance is also supported for channels.

Channel parameters can be assigned values when used in connections. Several connections can share a single "if" clause, and it will be possible to create and use temporary variables within loops; these features are especially useful for creating random topologies.

Expressions

Expressions may reference other parameters of the same module. String operations are supported (concatenation, substring, comparison, etc.). There is a possibility to distinguish between numeric types (i.e. integer vs double).

A higher performance can be achieved by compiling expression into native C++ expressions instead of generating interpreted reverse Polish representations (cPar-sXElem expressions).

Self-documentation

NED-2 will provide a way to generate documentation from NED files in an automated way. The solution is similar to Javadoc: special comments are used to denote documentation in the NED source. An example:

```
/**
 * Packet type used by the Dyna example
 */
message DynaPacket
{
    fields:
        short packetType;  /// possible values: connreq, data, …
        int srcAddress;    /// source address
        int destAddress;   /// destination address
};
```

As for software that can actually produce documentation, Doxygen [7] may be used; it will have to be extended to support NED.

XML and OMNeT++

XML Overview

XML is a universal data interchange format. XML can be viewed as “generalized HTML” – it has a similar syntax but arbitrary tags. XML has increasingly become a generic format for presenting structured information. XML has been created and standardized by the World Wide Web Consortium (W3C). Today, XML enjoys a huge popularity. There is an abundance of specifications built around and on top of XML. It is currently used for web-related and e-commerce purposes and as standard file format (e.g. UML’s XMI specification).

There are standard tools for processing XML. XML parsers are used to convert XML to an in-memory representation. Many XML parsers can also verify that an XML document fulfills certain rules described by its DTD or XML schema; such parsers are called validating parsers. A standard in-memory representation for XML is DOM – parsers can build DOM trees (DOM parsers) or let the user build their own data structure (SAX parsers). Stylesheet transformations (XSLT) can be used to convert XML documents into each other or into other file formats.

There is already an abundance of XML-related tools. To mention a few: XML-based database access and XML databases; visual tools for designing transformations between XML documents; tools for easy presentation of XML.

XML-based NED Architecture

For OMNeT++, XML is ideal as an alternative representation of NED. By supporting XML, NED becomes more interoperable with other systems, and it will be possible to process it with standard tools. XML is unsuitable though as the only representation of NED information because of its relatively pool readability.

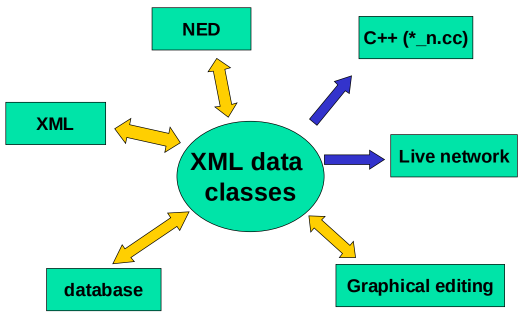
An example XML fragment that describes an OMNeT++ compound module:

```
<?xml version="1.0" ?>
<nedfile filename="fddi.ned">
    <module name="FDDINode">
        <params>
            <param name="address" datatype="string"/>
        </params>
        <gates>
            <gate gatetype="in" isvector="false" name="net_in" />
            <gate gatetype="out" isvector="false" name="net_out" />
        </gates>
        <submodules>
            <submodule name="MAC" typename="FDDI_MAC">
                <substparams>
                    <substparam name="mac_address" value="address"/>
                    <substparam name="promiscuous" value="false"/>
                </substparams>
            </submodule>
            ...
```

The planned NED-XML infrastructure is centered around data classes that are the in-memory representation of NED, and several components build around that. These components are:

- NED parser
- NED generator
- XML parser (incl. validator)
- XML generator
- C++ code generator
- network builder

The NED-XML architecture is illustrated in the following diagram:



Based on the infrastructure, a NED compiler can be assembled from a NED parser and a C++ code generator component. GNED can utilize the NED parser and NED generator components. Both nedc and GNED will be able to import and export XML by just adding the XML parser and XML generator components.

The infrastructure will make it easier to build models dynamically. For example, when building networks from data coming from a database, one might let the database query produce XML (several databases are already capable of that), then apply an XSLT transformation to convert to NED-XML if needed. Then one might apply the NED generator to create NED source; C++ code generator to produce code to be linked into the simulator; or the network builder to set up the network directly within the executable.

NED-2 Roadmap

The steps necessary for implementing NED-2 are thefollowing:

1. message subclassing (perl-based prototype already exists)
2. port existing nedc to XML architecture. This consists of writing the NED parser, XML data classes and C++ code generator. (Some code already exists.)
3. finalize NED-2 draft (feedback from the community is needed!)
4. complete NED-XML architecture for NED-2
5. nedc for NED-2

Implementation will be done by Andras Varga, based on feedback (and potential contribution) from the community.

Future plans include migrating GNED to NED-2 and creating a C++ network builder component that can be integrated into Cmdenv/Tkenv. The latter would enable new models without recompilation.

References

[1] OMNeT++ Discrete Event Simulation System. http://www.hit.bme.hu/phd/vargaa/omnetpp.htm

[2] NED-2 Draft specification. http://www.hit.bme.hu/phd/vargaa/omnetpp/neddraft.htm

[3] Generation Gap. C++ Report, Nov./Dec. 1996.

[4] Vlissides, John: Pattern Hatching: Design Patterns Applied. Addison Wesley, 1998

[5] Common Object Request Broker Architecture (CORBA/IIOP), version 2.5. OMG Specification.

[6] XML 1.0. W3C Recommendation, February 1998. http://www.w3.org/XML/

[7] Doxygen. http://www.doxygen.org

OMNeT++ Model Convergence: Aspects and Solutions

Dipl.-Ing. Ulrich Kaage

Universität Karlsruhe, Institut für Nachrichtentechnik, D-76128 Karlsruhe, Germany
kaage@int.uni-karlsruhe.de

Abstract. Since OMNeT++ is a simulation tool that is now used at a number of universities and companies, a lot of man power is spent on creating simulation models. However, as OMNeT++ does not offer a comprehensive model library so far, a lot of redundant work is done. It is obvious that models taken from different sources are not guaranteed to work together seamlessly. To ease the task of creating compatible models some programming guidelines and possible improvements to the OMNeT++ simulation environment are proposed in this paper.
These guidelines cover definitions of interface messages and documentation aids. New features of the upcoming OMNeT++ version make this task feasible and are discussed in detail.
Further proposed enhancements to the OMNeT++ simulation environment could lead to more hierarchically structured compound modules that can be inspected with OMNeT++'s graphical front-end TKENV.

1 Introduction

Taking an ever growing mailing list as an indicator for popularity, OMNeT++ [**?**] seems to start a career as a simulation environment for both research and development tasks in the widespread field of network and communications engineering as no day goes by without at least one or two queries about it. Success is usually based on more than one reason and so is OMNeT++. Reasons for its increasing popularity are that

– it is open source
– it is easy to learn
– it offers automatic visualization
– it offers graphical debugging aids
– its programming interface is well structured
– its framework allows models to be reusable

Since OMNeT++ development has been for a long time mostly a *one man's job* there are clearly some disadvantages, too:

– new features and enhancements sometimes find their way slowly into the simulation environment
– there is a lack of a comprehensive model library

While the first point is not always a disadvantage – it might even guarantee that the simulation kernel will not be bloated by too many additions – the second one, however, is indeed a big one. This is where the OMNeT++ user community comes into play.

There is a great variety of fields where simulation using OMNeT++ is applicable: From modeling processor architectures to Internet based protocols – as long as the system can be described as a discrete one – OMNeT++ might be the right choice. Anyway, there are actually some hot spots like mobile communication where a lot of research groups are currently working on. These groups would benefit from a comprehensive model library that provides well known and state of the art protocols. A list that is far from being complete is shown in table **??**. Note that protocols marked with an asterisk are either already available or currently under development. Unfortunately, these protocols are developed independently from each other and thus a combination of them into a more complex simulation is not guaranteed to work.

Layer	Protocol
Application	FTP, HTTP, RTP*, generic traffic generators and data sinks
Transport	TCP*, UDP*
Network	IP*, IPX, RSVP
Routing	OSPF, BGP, IGRP, RIP, EIGRP
Data Link	ATM, Ethernet, FrameRelay, LAN Emulation, TokenRing, FDDI, FibreChannel*, PPP, Wireless LAN, HiperLan*
Physical	SONET, xDSL, ISDN, Radio Link*

Table 1. Protocols

With the availability of such a library it would be possible to build complex simulation scenarios where protocol improvements or additions could be tested in interaction with other protocol layers. This would allow realistic performance evaluations with protocol interactions at different time scales being taken into account.

The big questions are: *How can a model convergence be achieved so that existing protocol models really become reusable and interchangeable? What kind of framework is necessary so that new developed protocol models fit into an existing structure?*

Some tasks have to be defined for these goals:

- define a model hierarchy where each protocol has its unique location
- define sets of interfaces that create links between different protocol layers
- recommend some tools for documentation of protocol models
- recommend programming guidelines that ease these tasks

The following sections will discuss these items in detail. Where necessary, future enhancements of the OMNeT++ simulation environment will be taken into account.

2 Model Hierarchy

Providing a comprehensive model library implicates some organizational preliminaries. One point clearly is its distribution approach. If model components are hosted at different sites and are therefore spread over the Internet it is a tedious task to collect the models of interest. Moreover, model improvements are likely to affect other models, too, that may be not known to the developer. Therefore, a unique location for models is advantageous.

A well known place is, e.g., SourceForge [**?**] that is owned by the Open Source Development Network, Inc. (*OSDN*). Today there are 31,393 hosted projects and 322,716 registered users. Moreover, this site offers discussion lists, IT news, development tools and a bug tracking system. On the downside, it has become a really slow site and often is not even reachable.

The *Institut für Nachrichtentechnik* at the University of Karlsruhe, Germany, has therefore begun to provide its own server that is dedicated to OMNeT++ development [**?**]. As a non-profit service this server is open to the public and offers a CVS-Repository for OMNeT++ models and the OMNeT++ sources. There are model specific web pages as well as mailing lists for communication purposes. Secured access controls allow to differenciate between privileged users that have write access to their models and unprivileged users that have anonymous read-only access to models that are made open source. This service is now running since November 2000 and has proven to be a reliable repository.

Developers of open source OMNeT++ models should feel encouraged to host their code on this server so that it becomes a unique resource for OMNeT++ development.

This would be the first step for collecting a model library. The next step should be to define a module library hierarchy that considers all models and links them together. However, care must be taken to assure that there will be not too many dependencies between models – a modular approach will be necessary. Unfortunately, more than one exists:

1. A model-based structure will result in a hierarchy that mostly pleases model developers since it is fairly easy to extend the model library by creating a new sub-directory that contains the new model. However, this flat structure could lead to a lot of redundant code being written.
2. An ISO/OSI protocol layer-based structure provides a hierarchy that reflects much more the protocol purpose. On the other hand, this could lead to models being spread over more than one sub-directory and therefore will increase the maintenance overhead.

Making a compromise, the directory layout could be structured according to table **??** which leads to a directory depth of at least two. One hierarchy for the protocol layer, one for the protocol model itself. Further sub-directories may be created if the model is of a great complexity.

3 Interfaces

Putting all protocol models into sub-directories will not automatically result in a modular interchangeable structure. Special care has to be taken of protocol interfaces. Starting with the lowest protocol layer, an interface message towards the upper layer should be defined that not only carries the protocol specific header and payload but also interface control information to control protocol layer specific behavior.

If the internal state of a module has to be made visible to other modules, a function call interface could be implemented that allows other modules to directly query and set the internal state or to exchange some information that does not fit into interface messages.

3.1 Abstraction Layers

At different protocol layer levels insertion of additional abstraction layers might be useful to provide a unique interface to upper layers. As an example taken from real implementations, the BSD socket layer provides an abstraction of the transport layer towards the application layer. Using sockets is a convenient way to hide transport specific functionality. Therefore, it becomes quite easy to write generic traffic generating applications that can be put on top of TCP, UDP or IP.

3.2 Message Subclassing

The easiest way to add protocol header information to a message is by using the `cMessage::addPar()` member function but this is deprecated for the following reasons. The most important one is that header parameters are generated on the fly in the source code. There is no clean header description that is visible to a user. Knowledge of the whole source code is needed for not missing an important header parameter. Moreover, parameters added by `addPar()` are referenced by a string value. Misnaming a string will not be caught at compile-time but at run-time. Tracking down run time errors is known to be a tedious task.

To compenstae this disadvantage, message subclassing should be used instead of `addPar()`. It is a much cleaner way of defining interface messages since all message information is contained in a class definition. Understanding the usage of a protocol module is therefore facilitated by looking at the message class header files.

Using message subclassing so far results in writing a lot of code: In addition to the protocol header fields getter and setter functions have to be written to access these. Furthermore, methods of the base cMessage class have to be redefined, i.e. constructors, assignment operator and some methods that interface with the OMNeT++ kernel.

OMNeT++ releases later than version 2.1 will offer a new way of defining `cMessage` sub-classes by means of a NED language extension introduced by András Varga. It will be no longer necessary to write code that defines a message subclass. Simply a NED file has to be created that contains the message definition. Compiling the NED file with nedc creates a header and implementation file. The latter one contains the implementation of the generated classes. Additionally, reflection code is added that allows to display message content with OMNeT++'s graphical front end TKENV.

3.3 Function Call Interfaces

As OMNeT++ is a discrete event-driven simulation environment, information exchange between modules happens by means of message passing. This is a classic approach that satisfies most needs. However, sometimes a function call interface between modules would ease the task to modularize a model design. This is true especially for global modules that are accessed by others to retrieve information.

The simulation environment already offers two mechanisms to retrieve a module pointer. One is implemented with the `cObject::findObject()` function call. The other one is by following gate connections until the module of interest has been reached. Both methods create access to a module class pointer and therefore it is possible to call its member functions.

This mechanism, however, is invisible to the user and might lead to simulation models that are hard to understand since access to other module pointers is only revealed by reading the appropriate part of the source code. This is where a function call interface might be the solution which could be declared within the NED module description. Its big advantage is that this kind of declaration could open ways for graphical front ends to display these module relationships by wires and somehow visualize module access.

4 Documentation Aids

One important step in writing reusable models that is often neglected is documentation. As a lot of models are programmed by students at universities doing their master degree documentation is often written as an external document. If the model is further actively developed, documentation usually falls behind. To circumvent this documentation inside the source code is really important.

Since OMNeT++ version 2.1 Doxygen [**?**] is used as the documentation framework for the kernel sources. It is an inline java-like documentation system for C++, Java, IDL and C.

Doxygen provides the following features:

- it generates an online documentation browser (in HTML) and/or an offline reference manual (in LaTeX) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, this facilitates to keep the documentation consistent with the source code.
- it can be configured to extract the code structure from undocumented source files. This can be very useful to quickly find one's way in large source distributions. The relations between the various elements are visualized by means of including dependency graphs, inheritance diagrams, and collaboration diagrams, which are all automatically generated.
- it can even be used for doing other kind of documentation

Using Doxygen therefore allows code documentation that stays up to date as the model evolves. However, Doxygen is not aware of the NED language files containing NED code. Extending Doxygen so that it knows how to parse NED code might be very useful.

5 Programming Guidelines

The topics described so far can be summarized into some programming guidelines that help make models written by different development groups being more consistent and therefore easier to be integrated with each other.

- find out if there is a model already available that can be extended to satisfy your needs.
- do not use `addPar()` to construct messages headers – use message subclassing instead.

– avoid using enums with a global scope. Instead, put them into the module or message class they belong to. This avoids name clashes and makes it easier to read the code.
– for each module or message use one class and one header file. Try to avoid to put together different classes into the same file. This is also true for NED description.
– use Doxygen for inline documentation of your simulation model.
– consider contributing your simulation model to the OMNeT++ CVS server.

6 Conclusion

This paper tried to point out some issues that are worth while being discussed in order to lay the fundamentals for a comprehensive OMNeT++ simulation class library. What has been proposed should be subject to further elaboration on the OMNeT++ mailing list with the goal of defining a concrete framework to support developers writing models that can be effortlessly integrated into the model library.

References

fN02. Institut für Nachrichtentechnik. Omnet++ repository homepage. `http://cvs-int.etec.uni-karlsruhe.de`, 2002.

OSDN02. Inc. Open Source Development Network. Sourceforge homepage. `http://www.sourceforge.net`, 2002.

Var02. A. Varga. Omnet++ discrete event simulation system homepage. `http://www.hit.bme.hu/phd/vargaa/omnetpp.htm`, 2002.

vH02. Dimitri van Heesch. Doxygen homepage. `http://www.doxygen.org`, 2002.

Planning New Features for the OMNeT++ Simulation Kernel

András Varga
Department of Telecommunications
Budapest University of Technology and Economics
andras@whale.hit.bme.hu

Abstract

This paper describes issues and planned new features for the OMNeT++ simulation kernel. Many of the items presented here are not (fully) elaborated yet, and the OMNeT++ community is strongly encouraged to provide feedback about them. The items listed here include: beautifying the simulation library (const-correctness**,** STL-style iterators, more inner classes, etc.); object-oriented random number streams; multi-thread modules; more flexible channel objects; real-time scheduler (for hardware-in-the-loop simulation); parallel simulation; generating ns2 nam traces; whether it would be useful to introduce scripting).

Two larger projects (about NED-2 and using Java as future tools platform) are described in separate documents. Parallel simulation is introduced in a paper from Gabor Lencse.

Beautifying the Simulation Kernel

The OMNeT++ [1] simulation kernel has undergone several changes since the 2.1 release. The C++ code has been made const-correct, and several changes were made to make the code more consistent and robust. Further changes are planned to make the code cleaner, for example introducing STL-style iterators and making several of the small auxiliary classes inner classes.

Object-Oriented Random Number Generators

Current solution for generating random numbers is the intrand() and genk_intrand() functions. Distributions are also implemented as functions that call intrand() and genk_intrand() directly. This solution is flawed in several ways. First, the total number of random number generators is limited, and the limit can only be changed by recompiling the simulation kernel. Second, distributions cannot be made to use other random number generator than the one built in. Third, the solution is not object-oriented.

The current solution has to be replaced by an object-oriented one that cures the problems mentioned above. A possible model is the GNU C++ library. The design is complicated by the requirements that one should be able to use the random number generators and distributions transparently from NED expressions, and also transparently be able to set seeds from the ini file.

Details are to be worked out.

Multi-thread modules

Currently, if there is a need to have multiple threads of control within a simple module, dynamically created modules have to be used. It is usually difficult to have these modules share resources (i.e. have access to a shared queue or status variable). A more appropriate solution would be to support multiple threads of execution within a simple module; the main advantage would be that a “thread” could use resources (data members like queues, etc.) of the containing simple module.

A rough initial design is the following. Introduce a **cProcess** class that is a unit of scheduling and execution (“lightweight simple module”). It would have the following properties:

- An event’s target is always a cProcess.

- cProcess has activity() and handleMessage()

A cSimpleModule is (has?) a cProcess. A simple module may create/destroy further cProcesses. The child cProcesses live within the parent module's context (may access its parameters, gates, members, etc.)

Messages from input gates are always received by the cSimpleModule (its main cProcess), which may in turn manually dispatch them to its cProcesses. A cProcess may have its own scheduled self-messages which are delivered directly to it.

cProcess might have uses outside simple modules too, potentially in channels. (The channel concept is still to be created.)

Channels

Motivation

The current channels provided by OMNeT++ are very simple which limits their usefulness. Problems with current channels:

- no custom attributes
- not scheduled → anomalies with animation and sometimes with model logic
- not customizable (simplistic error model, awkward busy channel modeling)

A more flexible channel model is needed. To be able to design a better solution, we need to explore the requirements. The fundamental question is the following:

> **Complex channels could be implemented using simple modules. Do we need powerful and flexible channels or are basic channels enough (and use simple modules for complex channels)?**

The question is open to discussion. This paper doesn't attempt to answer it, just lists some points that can help in making the decision.

Design issues

The right solution depends on what we want to use channels for. What do we want to model using channels?

- point-to-point? shared media? bus?
- delays? error model? physical propagation models (radio)? collisions?
- complex channels vs. simple channels+modules

Considerations for the design:

- do we need scheduling (events) for modeling transmission? → YES (current implementation must be changed!)
- should the design allow several connecting channels in a path? → NO (otherwise checking channel state from simple modules becomes complicated. Current implementation must be changed!) -> we must distinguish between simple connections (no delay/error/datarate modeling) and channels
- channels will need arbitrary parameters (like modules)
- do channel implementations need to be modular/hierarchical? → yes? maybe?
- do channels need several gates? (for bus or shared media) → no? (introducing named gates for channels would also cause notation problems in NED)

- what is the technical difference between such complex channels and modules? → different semantics but very similar implementation?

Two (three) ways seem tractable for the channel design:

- Lazy: support only very simple channels (remove BER and datarate attribute) → then more complex channels should become simple modules.
- Heavyweight: build a mechanism parallel to modules for channels (parameters, maybe gates, activity()/handleMessage(), etc.)
- Middle: add channel parameters; channel object is a cProcess (can be written by user)

Feedback is kindly requested from the user community.

Hardware-in-the-loop simulation

Hardware-in-the-loop simulation enables interesting scenarios that are sometimes of huge practical value:

- The simulator may represent part of the real network.
- A real-world device may be tested inside a simulated network.

Such scenarios naturally assume that the simulation program is running on a computer that is fast enough to be able catch up with events in the real life.

It is planned to enhance OMNeT++ to support hw-in-the-loop simulation. The needed simulation kernel improvement is a real-time scheduler. The interface for communication between the simulation and the real system is usually specific to the concrete simulation scenario and is therefore difficult to generalize.

Having a real-time scheduler in addition to the normal scheduler raises the need that the scheduler inside OMNeT++ be replaceable.

Implementation of the real-time scheduler has already begun (Andras Varga). However, it is currently suspended because of practically more important tasks (e.g. NED-2 [2]).

"nam" traces

The ns2 [3] project's nam program is a very useful off-line network animation viewer. OMNeT++ currently lacks such a tool. If OMNeT++ could generate ns2 nam traces, the OMNeT++ community could make use of nam instead of having to write another animation viewer.

Using nam is feasible because:

- it already has (most) features we need
- available (open-source, etc.)
- works on a well-documented, simple input file
- fairly independent of ns2

The task is to enhance OMNeT++ and make it able to generate nam traces. This can be done by enhancing the cEnvir library; details about the nam trace can be provided via the omnetpp.ini file.

The task has been handed out as a student project at TU Budapest by Dr. György Pongor; work is still in progress.

Scripting

Motivation: ns2 uses Tcl to script simulations. OMNeT++ currently does not have a scripting facility. Tasks that are solved in ns2 by simulation scripting are solved in OMNeT++ in other ways: via module parameters that obeyed by simple modules; custom script files understood by individual simple modules; or C++ programming.

The question is whether OMNeT++ needs simulation scripting. Pro and con arguments are the following.

We do *not* need scripting to:

- build up topology (done via NED)
- assign values to parameters (done via ini files)

We *might* need scripting to:

- express dynamic scenarios (where traffic pattern changes over time, nodes/links go down and up, etc.)

Opinions are expected from the OMNeT++ community.

OMNeT++ Distributions

Several OMNeT++ binary distributions are/should be maintained.

There were efforts to create distributions for various Unix systems.

- Linux RPM package
- Debian package
- Solaris package

Regarding Windows support and MSVC integration, the following components have been created:

- OMNeT++ AppWizard
- “Add NED file” tool
- Installer (NSIS-based)

Maintainers are needed for all the above components.

References

[1] OMNeT++ Discrete Event Simulation System. http://www.hit.bme.hu/phd/vargaa/omnetpp.htm

[2] NED-2 Draft specification. http://www.hit.bme.hu/phd/vargaa/omnetpp/neddraft.htm

[3] The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns/

Session 5: Last minute presentations

A number of presentations have been announced after the deadline for the camera-ready version had expired so that no article could have been included. They are:

- Simulated-KIDS — A flexible simulation suite for individual Quality of Service mechanisms, *Klaus Wehrle (Universität Karlsruhe (TH))*
- Adaptation of a router simulation to realistic delays, *Eckehardt Luhm (Universität Karlsruhe (TH))*
- A flexible traffic generator for realistic Internet traffic, *Stefan Sellschopp, Milena Neumann (Universität Karlsruhe (TH))*