Proceedings of the 2004 Winter Simulation Conference
R. G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, eds.

INTEGRATION OF THE FREEBSD TCP/IP-STACK INTO THE DISCRETE EVENT SIMULATOR OMNET++

Roland Bless
Mark Doll

Institute of Telematics
University of Karlsruhe
76128 Karlsruhe, GERMANY

ABSTRACT

The discrete event simulator OMNeT++, that is programmed in C++, shows a steady growing popularity. Due to its well-structured nature, it is easy to understand and easy to use. A shortcoming of it, however, is the limited number of available simulation models. Especially, for network simulations a validated TCP implementation was missing. In order to avoid a re-implementation of a full-featured TCP, including all potential implementation errors and costly validation tests, we integrated a TCP/IP stack of a real operating system into OMNeT++. In this paper we show that such a port is feasible with reasonable effort and we describe difficulties of the integration process as well as the applied solutions. We also present some evaluation results that outline memory and CPU usage.

1 INTRODUCTION

Investigation of new network protocols and mechanisms often require simulations in order to study their behavior and reactions to different parameter settings in larger scale environments. But results are only reliable in case the simulation models are verified to emulate the real protocol behavior. Therefore, *validated* implementations of network protocols in simulators are an important precondition for meaningful simulations.

OMNeT++ (Varga 2004) is a discrete event simulator based on C++, and it is highly modular, very well structured and scalable. It provides a basic infrastructure wherein modules exchange messages. Thus, it is not restricted to network simulations and due to its hierarchical structure, the level of simulation detail can be increased incrementally.

Its disadvantage, compared to other simulators, like ns-2 (Information Science Institute (ISI) 2004) for example, is the currently limited amount of available simulation models for different network protocols and technologies. One of the predominant protocols in the Internet is the *Transmission Control Protocol TCP*, because most applications, including the World-Wide Web transfer protocol HTTP, are using it for reliable data transfer. But for the discrete event simulator OMNeT++ there were no validated TCP implementations available yet. The TCP implementation must be complete, i.e., it must offer all essential features that real implementations use. Moreover, its behavior must also be compliant with the relevant standards. Therefore, many tests must be performed to validate the implementation against all protocol features which is costly and time consuming. A lack of validation, however, may lead to wrong simulation results and conclusions.

Our approach aimed to re-use an existing full-featured TCP implementation of a current operating system in order to avoid a re-implementation of a full-featured TCP, including all potential implementation errors and costly validation tests. We adapted the FreeBSD TCP/IP implementation to integrate it into OMNeT++. We choose FreeBSD (FreeBSD 2004), because its implementation is more structured than other implementations like Linux. The efforts were indeed successful so that now a full featured TCP/IP implementation is available for OMNeT++.

The paper is organized as follows. First, we describe some properties of the OMNeT++ and FreeBSD respectively. Then integration problems of FreeBSD and OMNeT++ are pointed out in section 3. The applied evaluation process is described in section 4. The paper closes in section 5 with a summary and outlook on further work.

2 OMNET++ AND FREEBSD

OMNeT++ (Varga 2004) is a discrete event simulator programmed in C++. The name OMNeT++ stands for *Objective Modular Network Testbed in C++*. It has an open-source distribution policy and can be used free of charge by academic research institutions. It runs on Windows and Unix platforms, including Linux, and offers a command line interface as well as a graphical user interface. In this paper, we

focus on network protocol simulation, although OMNeT++ can be used, for instance, to model queueing networks, multiprocessors and other distributed hardware systems as well as to validate hardware architectures, too.

Its simulation models consist of a network of *simple modules* and *compound modules*. Due to the fully hierarchical design of OMNeT++ the latter can be composed of simple modules or further compound modules (there is no limit of nesting levels). The *system module* is the top-level module that encompasses all compound and simple modules of a simulation. An example for a module hierarchy is shown in Figure 1. Simple modules are programmed in C++, the network topology and simulation parameter values are specified in an own language that is called NED (Network Description). A module can have *gates* that establish connections to other modules via *links*, which have an assigned data rate and bit error rate. *Messages* between modules are either sent via gates and traverse the outgoing links, or, can be sent directly to other modules. Simple modules typically 'wait' for messages which can also stem from the module itself (so-called self messages). Upon arrival of a message the module can perform the necessary actions in a method called `handleMessage()` (processing within this method takes no simulation time). The module usually generates new messages and sends them to other modules or itself. These messages are inserted into a central queue (the so-called *future event set* that is implemented as heap) and taken out by the receiving module. Simulation time only passes from event to event in discrete steps, so OMNeT++ has its own time domain.

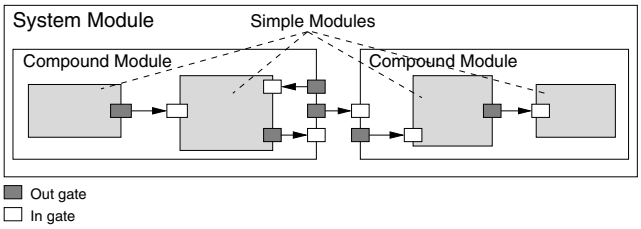

Figure 1: Example of a Module Hierarchy in OMNeT++

For network simulations ns-2 is a well-known discrete event simulator targeted at networking research. It provides a huge number of different protocol models, e.g., simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks. On the one hand, if one compares OMNeT++ with ns-2 one may find that OMNeT++ has a cleaner structure (it is thus easier to learn), and, that it is more scalable (Bless 2002). On the other hand, there were no validated TCP implementations available for OMNeT++. In order to find a remedy for this problem, we wanted to integrate a TCP protocol implementation of an existing and widely-used operating system into OMNeT++. This should avoid a re-implementation of a full-featured TCP, including all potential implementation errors and costly validation tests.

FreeBSD is a freely available UNIX implementation and its kernel offers an implementation of TCP. FreeBSD has a traditional monolithic kernel and uses function calls as well as interrupts to perform its tasks. For instance, such an interrupt is triggered by a network interface card (NIC) that receives a data packet. At the lowest layer, a device driver for the NIC allocates a memory buffer, a so-called *MBuf*, and copies the packet contents into this buffer.

Table 1 shows a comparison of the different features. In order to use functions of FreeBSD in OMNeT++ we have to overcome several differences. The integration process is described in the next section.

3 INTEGRATION OF FREEBSD INTO OMNeT++

In order to accomplish the integration of FreeBSD into OMNeT++ we need to find a synthesis between the distinct features of the different worlds. The monolithic nature of the FreeBSD kernel is no real obstacle, because integration of the complete code into one module may be easier this way. Because C is a subset of C++, it is also possible to use C code with C++ code together, but one must take care of the different linkage.

Table 1: Overview of Major Differences between FreeBSD and OMNeT++

Feature	FreeBSD	OMNeT++
Structure	monolithic	modular
Scope	one TCP stack per host	several TCP stacks per simulation
Interaction	function calls	message sending
Interruption	hardware/timer interrupts	not possible while processing a message
Language	C	C++

3.1 Source Code Adaptation

One problem were the conflicting definitions of similar include files within the OMNeT++ host system (in our case Linux) and FreeBSD. For example the structure `sockaddr` shows slight differences between Linux and FreeBSD that are nevertheless incompatible with each other. The solution was to use all the FreeBSD include files for compilation of the FreeBSD code and to define corresponding structures with a different name on the OMNeT++ side. This has also the advantage that the FreeBSD part may be also easily ported to the version of OMNeT++ that runs on top of the Windows operating system. Thus, OMNeT++ uses the include files of its host operating system, and the FreeBSD

part uses its own set of include files to use the correct structure definitions.

The main problem to solve was related to the scope of variables. In FreeBSD kernel variables are globally declared and defined for one host only. But in a simulation environment one wants to run several hosts in parallel, so every required global and static FreeBSD variable must be made local to each host. Thus, we used a structure to store all the kernel variables for one host. Consequently, in the FreeBSD source every occurrence of the variables must be replaced by a reference into the corresponding structure. For instance access to the variable `xyz` is replaced by `D->xyz`, where `D` points to the current host structure that contains all the global and static kernel variables for this particular host. Unfortunately, the developers of FreeBSD used at several places the same name for a variable and a type, e.g., the variable `ifnet` is also defined as a structure type with the same name. Thus, it was not easily possible to use a straightforward search and replace approach by a simple script. However, changes in the FreeBSD were minimized in order to reduce the probability of introducing new errors and to allow for easier re-porting a later FreeBSD release.

3.2 Modules and Interactions

Early in the adaptation process it became clear that it was much easier to adapt and integrate also functions like ARP and the whole IP stack to OMNeT++ than to provide those functions in OMNeT++ itself. This is reasonable due to the existing dependencies of several protocols. For example, to support path MTU discovery for TCP, the Internet Control Message Protocol ICMP must be implemented, too.

Our design decision led to the approach to encapsulate the complete TCP/IP into one OMNeT++ simple module. Thus, the `cHost` class and OMNeT++ module encapsulates the complete TCP/IP stack of FreeBSD and offers a message-based interface to the application as well as an interface (in- and out-gates) to the medium (cf. Figure 2). From the FreeBSD's viewpoint OMNeT++ is like a device that transmits and receives ethernet frames.

A disadvantage of this approach may be that a FreeBSD kernel cannot be removed easily, because the code is not written for a proper cleanup since it is usually makes no sense to remove a kernel.

There are two possibilities how hosts are connected to each other: either by a direct point to point link (which is also handled separately in FreeBSD) or via a broadcast capable medium. Because OMNeT++ does not support 1:*n*- or *n*:1-connections directly, one must provide a separate module for this purpose. This module is provided by the class `cMedium` which has basically the same functionality like a simple ethernet switch. It is possible to assign IP network prefixes to the `cMedium` in order to configure IP addresses of attached hosts automatically.

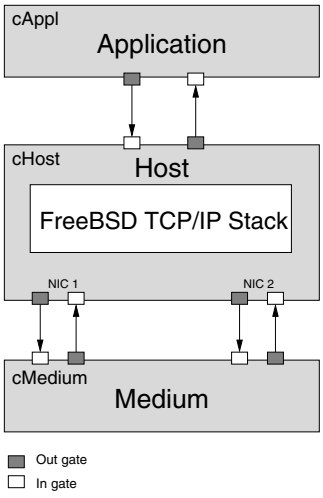

Figure 2: Module Structure

OMNeT++ messages cannot be used to exchange data between both worlds, because they are C++ classes and cannot be used within the FreeBSD part. In a `cHost` module data can be put from OMNeT++ into the FreeBSD domain by calling the FreeBSD function surrounded by `ENTER_BSD` and `LEAVE_BSD` macros. The other direction is more complicated. Functions that can be called from the FreeBSD domain must be present in the `cHost` class, but this is C++ code and cannot be called from the C-based FreeBSD part directly. Therefore, those functions must be declared as functions with C calling conventions in the OMNeT++ part. These can then access the current `cHost` C++ object by using a `this` pointer that was set correctly by the last `ENTER_BSD` macro. This is guaranteed to work, because the FreeBSD part only gets active when called from within OMNeT++ `cHost` class and after an `ENTER_BSD`.

The application interface for using TCP is essentially the same as the well-known socket interface. The difference is that the traditional function calls are not used, but equivalent OMNeT++ messages that are sent between both modules (cf. Figure 2). Every socket function has an equivalent OMNeT++ message that simply contains all the necessary function parameters. In the `cHost` module C wrapper functions convert between OMNeT++ messages and function calls. Results of FreeBSD function calls are also returned by messages to the application. The `cAppl` class provides a sample application that simply opens a connection to another host and sends a specified amount of data. This way it can be used as load generator. The data can contain real application data, because the application byte stream is really passed from one host to the other.

A further adaptation problem occured with function calls that block the calling process, or with functions that let a process sleep or wait for a while. They must not stop in the FreeBSD kernel part, because then the simulation would also stop. This is due to the fact that a `handleMessage()`

procedure must finish in order to return control to the OMNeT++ simulation kernel, so that it can pick the next message from the event queue. Therefore, we used internally (in the `cHost` module) only non-blocking variants of the FreeBSD calls, but provided also blocking variants to the application interface. This works as follows: if a FreeBSD function would normally block the caller, it returns the error code `EWOULDBLOCK` when called as non-blocking variant. In this case the `cHost` module does not send a return message, but stores a pointer to the original request message. If the kernel would wake up the caller, the list of 'sleeping messages' is searched instead. If the corresponding message is found, it is sent again (as self message) to the `cHost` module. This ensures that all necessary actions are taken and that changes in the internal state of the kernel are considered. But this time the function call will definitely not return an `EWOULDBLOCK` and it can be completed as usual.

3.3 Timers

Timers were another problem. The FreeBSD stack needs several different timers per host (e.g., `if_slowtimo`, `arptimer`, `in_rtqtimo`, `tcp_slowtimo`, `ip_slowtimo`) and per TCP connection (e.g., `tcp_timer_rexmt`, `tcp_timer_persist`, `tcp_timer_keep`, `tcp_timer_2msl`, `tcp_timer_delack`). In order to reduce the number of timers, we used only single timers were appropriate, e.g., the `ip_slowtimo` timer simply deletes fragments that were not reassembled within a defined time period.

The time basis is the simulation environment, so the (virtual) clock must reside in the OMNeT++ part. FreeBSD uses a 'tick' as time unit which is 10 ms, and the kernel global variable `ticks` is incremented every 10 ms by a timer interrupt routine. In order to prevent a lot of OMNeT++ messages, we did not choose to emulate the timer interrupt by OMNeT++ messages. Instead we redirected access to the `ticks` variable to a function `gettick_toomnet()`, which returns the number of hundredth of a second since the simulation ran. The OMNeT++ function `simtime()` returns the simulation time for this purpose, but it is a floating point value (double) and must be converted accordingly into the integer value. To prevent that all hosts increment their ticks at the same instant, a `startup` value is added to each `cHost` module, which can be set individually for each host.

Timers are set by specifying a time period as number of ticks. Therefore, we provided functions on both sides, to allow a timer management that is usable by the FreeBSD kernel. Basically, a timer is realized as self message in OMNeT++ that calls corresponding FreeBSD functions when it is received by the `cHost` module.

3.4 Convenience Functions

In order simplify the simulation configuration an automatic routing is provided by the OMNeT++ class `cRoute`. It uses an OMNeT++ internal mechanism to calculate the shortest paths between the hosts. Routers can be easily provided by switching the variable `bsd_ipforwarding` of a `cHost` module to 1.

For debugging purposes TCP traces are also provided. They can be enabled by setting the OMNeT++ simulation parameter `showtraffic` for the `cHost` module. Figure 4 shows an example output from the test scenario in Figure 3.

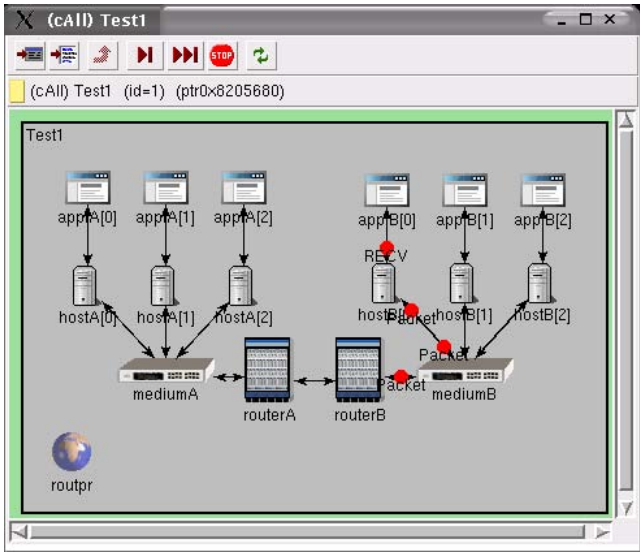

Figure 3: Snapshot of OMNeT++ GUI Showing a Simple Test Scenario

4 EVALUATION

The evaluation of the taken approach had two different goals. First, it was of particular interest to evaluate the memory and CPU consumption of the `cHost` modules to estimate the scalability of the model. Second, it was important to roughly verify the behavior of the TCP implementation to exclude any implementation mistakes.

All tests for the evaluation were performed on a Xeon dual processor system running at 2.2 GHz with 4 GiB RAM (1 GiB= 1024 MiB, 1 MiB=1024 KiB, 1 Kib= 1024 bytes).

4.1 Scalability

We ran tests with 10, 100 and 1000 hosts (plus additional 20 hosts that act as routers) and up to two simultaneous connections per host. The average memory consumption per host was determined to be around 20 KiB. Additional

```
->Test1.hostA[0] 2.000000 ARP (ARPHeader) REQUEST: 66:06:0c:b6:30:58 00:00:00:00:00:00 192.168.0.1 192.168.0.4
->Test1.hostA[0] 2.040140 TCP (TCPHeader) [3014554104...3014554104) (0)@0 win 57344 <SYN> MSS 1460 WSF 0 TS 200 0
->Test1.hostA[0] 2.200873 TCP (TCPHeader) [3014554105...3014554105) (0)@1491920413 win 57920 <ACK> TS 220 246
->Test1.hostA[0] 2.200928 TCP (TCPHeader) [3014554105...3014555105) (1000)@1491920413 win 57920 <ACK,PUSH> TS 220 246
->Test1.hostA[0] 2.325622 TCP (TCPHeader) [3014555105...3014555553) (448)@1491921414 win 57920 <ACK> TS 232 258
->Test1.hostA[0] 2.448015 TCP (TCPHeader) [3014555553...3014557001) (1448)@1491921414 win 57920 <ACK> TS 244 271
->Test1.hostA[0] 2.449277 TCP (TCPHeader) [3014557001...3014558449) (1448)@1491921414 win 57920 <ACK> TS 244 271
```

Figure 4: Example Debugging Output from a Simple Test Scenario

memory of 150–170 KiB is required per (bi-directional) TCP connection, which is mainly caused by the socket buffers.

Run times of the simulation experiments are shown in Table 2. Every simulation ran for a simulated time of 60 minutes. As one can see, initializing the 1000 hosts costs already a little bit more than a minute. To simulate 1000 hosts with a total of 2000 simultaneous connections for 1 hour simulated time, it took nearly 14 minutes of runtime. In this case, the simulation was four times faster than the same scenario in real time. We identified that most of the run time is required to process timer messages. OMNeT++ uses a heap to store messages in its future event set, so this can be considered as optimal. Nevertheless each message requires memory and must be inserted and removed from the heap. It is obvious that we will try to remove this bottleneck in the future.

Table 2: Run Times in Seconds for 1 Hour of Simulated Time

Simult. Connections / Hosts	0	1	2
10	0.467	2.199	4.196
100	3.361	30.575	59.638
1000	64.233	434.724	823.019

4.2 TCP Validation

We also performed tests in order to check that all TCP features work as expected. The number of tests was small, because we did not modify the TCP code of FreeBSD, so we did not have to test all potential error cases and TCP's reaction to them. The major motivation for using an implementation of a real operating system was to particularly avoid massive and thorough validation tests. Nevertheless, we shortly checked that mechanisms like delayed acknowledgments, slow start, congestion control, fast retransmission, and fast recovery as well as the new reno variant of TCP worked as expected. The short validation tests revealed indeed a bug that was introduced by a wrong timer offset calculation within `cHost` .

5 CONCLUSIONS AND FUTURE WORK

In this paper we described an integration of a real TCP/IP stack (FreeBSD) into the discrete event simulation environment OMNeT++. The main objective was to provide a validated TCP implementation for OMNeT++. The implementation provides a host module that carries a complete TCP/IP stack in it. Host modules can be used as routers, too, by simply activating the forwarding functions of FreeBSD. For convenience of simulation users the implementation provides an automatic IP addressing of modules and calculation of routing tables, too.

It was shown by a careful evaluation that the approach is scalable and works correctly. The approach has also the advantage to let one use the same code within the simulation environment as well as within a real implementation. Finally, there may be other discrete event simulators that could use the same approach.

In comparison to *emulation* approaches like User-Mode Linux (User Mode Linux Community 2004) or vBET (Jiang and Xu 2003) we believe that our approach is more scalable for network simulations, because it has not the full functionality of a complete operating system.

Currently, we are porting FreeBSD 4.9 and the KAME extensions to OMNeT++ in order to have a full-featured IPv6 and MobileIPv6 implementation. A perl script that uses a syntactical analysis to perform the replacement of (ambiguously named) variables in the BSD source code would be a great help here and is a topic for further research. This would also allow to quickly update the FreeBSD part for bug fixes.

Furthermore, we are investigating solutions to avoid using OMNeT++ messages for host timers. A potential solution could be the use of a dedicated timer module which manages timers on basis of ticks and which is able to perform direct callbacks into the FreeBSD code. We plan to release the TCP/IP stack to the public. Furthermore, in order to allow the use of routing daemons for simulating routing protocols, we want to port the necessary system call interface functions.

ACKNOWLEDGMENTS

We would like to thank Jérôme Freilinger who performed all the programming and evaluations.

REFERENCES

Bless, R. 2002, January. Using Realistic Internet Topology Data for Large Scale Network Simulations in OMNeT++. In *2nd Interna-*

tional OMNeT++ Workshop. Available online via <http://doc.tm.uka.de/2002/> file omnet_ws_2002-1.pdf [accessed August 20, 2004]. Technical University Berlin, Germany.

FreeBSD 2004, April. The FreeBSD Project. Available online via <http://www.freebsd.org/> [accessed August 20, 2004].

Information Science Institute (ISI) 2004, April. The Network Simulator ns-2. Available online via <http://www.isi.edu/nsnam/ns/> [accessed August 20, 2004].

Jiang, X., and D. Xu. 2003, August. vBET: a VM-Based Emulation Testbed. In *Proceedings of the ACM SIGCOMM 2003 Workshops*, 95–104. ACM.

User Mode Linux Community 2004, April. User Mode Linux Community Site. Available online via <http://usermodelinux.org/> [accessed August 20, 2004].

Varga, A. 2004, April. OMNeT++ Community Site. Available online via <http://www.omnetpp.org/> [accessed August 20, 2004].

AUTHOR BIOGRAPHIES

ROLAND BLESS is a senior research assistant at the University of Karlsruhe, Institute of Telematics. He studied Informatics at the University of Karlsruhe and got his Ph.D. degree Dr.-Ing. in February 2002. His research interests are Quality-of-Service, QoS management, Differentiated Services, Multicast, Mobility and QoS Signaling. He is actively participating in IETF Working Groups and brought parts of his work into the IETF standardization process. Dr. Bless is also member of IEEE and the German GI.

MARK DOLL is a research assistant at the University of Karlsruhe, Institute of Telematics. He studied Physics at the University of Braunschweig and joined the University of Karlsruhe in 2001. He is a Ph.D. student and has research interests in signaling and management of resource allocations for the Differentiated Services framework, especially in the case of multicast scenarios.