Integration of a GIST implementation into OMNeT++

Roland Bless
Institute of Telematics
Karlsruhe Institute of Technology
Zirkel 2, D–76128 Karlsruhe, Germany
bless@kit.edu

Martin Röhricht
Institute of Telematics
Karlsruhe Institute of Technology
Zirkel 2, D–76128 Karlsruhe, Germany
roehricht@kit.edu

ABSTRACT

The General Internet Signaling Transport (GIST) protocol was specified by the IETF in order to provide a generic transport protocol for signaling messages. A group at the Institute of Telematics implemented the GIST protocol and evaluated it already in smaller testbed setups. An evaluation of GIST in large-scale scenarios can, however, only be accomplished by using a simulation framework. In this paper we describe how the existing Linux-based and multi-threaded NSIS-ka implementation was ported to the OMNeT++ simulation framework. First we provide an analysis of the different design principles used and then describe related challenges. Then we describe a methodology for integrating an existing real protocol implementation into OMNeT++. The feasibility of the chosen approach is finally demonstrated by a set of evaluations.

Categories and Subject Descriptors

C.2.6 [**Computer-Communication Networks**]: Internetworking—*Standards*; I.6 [**Simulation and Modeling**]: Miscellaneous

General Terms

Measurement, Standardization

Keywords

OMNeT++, Simulation, NSIS, GIST

1. INTRODUCTION

The *General Internet Signaling Transport* (GIST) protocol was introduced by the Internet Engineering Task Force (IETF) as a generic transport protocol for signaling messages within the Next Steps in Signaling (NSIS) framework. GIST was designed to be used by a variety of different signaling applications, e.g., for performing resource reservations to provide Quality-of-Service or for configuring NAT-Gateways and Firewalls. All specified parts of the NSIS framework were implemented by the Institute of Telematics within the last couple of years. The C++-based implementation was subject to different interoperability tests and was evaluated in testbeds as well as within Internet-wide scenarios using Linux and standard PC hardware.

However, a detailed evaluation of network protocols in setups with several thousands hosts and within different sets of topologies cannot easily be accomplished by using real OS-based implementations running on real hardware due to cost and management constraints. Instead, *simulation frameworks* are usually used to test protocol implementations in scenarios with a large number of interacting hosts. In order to test and evaluate new network protocols, an implementation is typically designed for the simulation framework itself before an implementation for real hardware is subsequently carried out. OMNeT++ [7] is a generic discrete event simulation framework that is widely used for evaluation of network protocols.

In order to evaluate GIST also in large-scale setups with hundreds to thousands of hosts, the existing protocol implementation had to be ported to the OMNeT++ simulation framework. Rather than writing a new implementation for the simulation environment from scratch, we wanted to reuse as much of the existing code in an unmodified manner. The existing software is, however, modularized and heavily uses Posix threads as well as a self-written Linux-specific protocol library which offers transport services, timers, and useful data structures. The different design principles used by the GIST implementation and OMNeT++ pose a set of different challenges. In this paper we give a short introduction into the background and related work in Section 2. A detailed analysis of both implementation's relevant parts and the proposed methodology to resolve arising challenges are discussed in Section 3. In Section 4 we present some performance evaluations of the resulting integration before we conclude in Section 5.

2. BACKGROUND AND RELATED WORK

The Next Steps in Signaling (NSIS) framework [1] was introduced by the IETF in order to provide a generic signaling protocol suite in IP-based networks and in response to some deficiencies of the already present Resource Reservation Protocol (RSVP). Hence, NSIS follows a two-layered approach by separating the signaling application's logic from the signaling message's transport. The lower layer, called NSIS Transport Layer Protocol (NTLP), is responsible for routing and transport of signaling messages and makes use

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OMNeT++ 2010 March 15–19, Torremolinos, Malaga, Spain.
Copyright 2010 ICST, ISBN 78-963-9799-87-5.

of underlying transport protocols, such as UDP, TCP, TCP with TLS or SCTP. The GIST protocol [6] fulfills the requirements of an NTLP.

An implementation of the GIST protocol specification was developed at the Institute of Telematics [2] and was evaluated by a couple of interoperability tests against various other GIST implementations. The NSIS-ka implementation of GIST currently comprises about 34,000 lines of code which are mostly C++-based.

Porting existing implementations into simulation environments such as OMNeT++ was already subject of earlier work. A prominent example was the integration of FreeBSD's TCP/IP stack into the simulation environment [3, 5]. Some of the problems within this work were similar to the ones we encountered by porting GIST to OMNeT++ such as the scope of variables which are declared globally in FreeBSD's implementation whereas multiple instances of these variables must be used for the simulation of several hosts in parallel. However, not all of the proposed paradigms were applicable for our work. For instance, in contrast to FreeBSD's C-based implementation which is single-threaded, GIST's C++-based implementation heavily uses POSIX threads. Furthermore, the existing GIST implementation could not be encapsulated into one single OMNeT++ SimpleModule.

Different aspects of integrating real-world applications into OMNeT++ are examined in [4]. In OMNeT++ three different approaches towards an integration of existing applications are conceptually provided: socket connections, source code integration, and shared library integration. Challenges concerning such an integration—e.g., handling application threads in a discrete event simulator like OMNeT++ or transforming application-specific timer mechanisms—are outlined in this work. However, the proposed solutions are primarily exemplified for the shared library approach and are therefore only applicable in parts to our approach where we aim at integrating the source code of an existing implementation into OMNeT++ (version 4.0).

3. INTEGRATING GIST INTO OMNET++

According to the design of the NSIS framework the existing NSIS-ka implementation was separated into different components, a protocol library (*Protlib*), the GIST layer, and each NSLP application amongst others. In the context of this paper we focus on the GIST implementation and the one for the protocol library. Each of these components is subsequently separated into NSIS modules, e.g., a timer module or a state module (c.f., Figure 1). The C++-based implementation follows a multi-threaded approach that is technically realized by POSIX threads so that each NSIS module is subclassed by a `Thread` class and can comprise several running threads. Thread processing may be either implicitly controlled by the scheduler or one can explicitly synchronize them by letting them wait on specific condition variables. Concurrent access to shared data structures must be controlled by means of locks and mutexes. Communication between threads is achieved by using *FastQueues*, that are shared memory message queues using mutexes and condition variables to resolve concurrent access. A thread typically listens to one or more assigned FastQueues waiting for incoming messages sent by other threads or own messages. An NSIS module is initialized by a class constructor which retrieves its necessary parameters via a subclassed `ThreadParam` data structure.

OMNeT++ is a discrete event simulation framework that models message senders and receivers as subclasses of `cSimpleModules`. All internal messages are inserted into a *Future Event Set* and upon reception of a message the receiver's `handleMessage()` function is called by the simulation kernel. The constructor used to instantiate subclassed `cSimpleModules` must be empty. Therefore, initialization parameters cannot be passed via the constructor itself and an `initialize()` function must be used instead which retrieves the parameters usually via the `omnetpp.ini` configuration file or the module's specific `ned` file.

The relevant differences between the existing NSIS-ka implementation and the OMNeT++ simulation environment outlined in this section are summarized in Table 1.

	NSIS-ka	**OMNeT++**
Active entity	Thread	SimpleModule
Processing Mode	Parallel	Sequential/non-preemptive
Scheduling	Indirectly via thread conditions and synchronization	Directly on message arrival
Event signaling	Condition variables	Messages

Table 1: Comparison of the NSIS-ka implementation and the OMNeT++ simulation environment

Regarding the integration of NSIS-ka's GIST implementation into the OMNeT++ simulation environment, the different mechanisms being used lead to a diverse set of challenges. An important requirement was to keep as much as possible of the existing protocol implementation unmodified in order to allow future versions of the implementation to make use of this extension as well without having to perform considerable adaptations.

3.1 Modelling of cSimpleModules

An important design decision was related to the question which part of the existing implementation had to be modelled by a cSimpleModule. Modelling POSIX threads as cSimpleModules is advantageous regarding the logical coherence of entities receiving and emitting messages. However, in order to simulate conditions that trigger POSIX threads an additional message must be provided. Furthermore, this design does not result in an efficiency gain as the cSimpleModules are always processed sequentially by the simulation kernel and the high number of objects imposes a much higher memory usage which may lead to scalability problems.

For these reasons, we decided to model an entire NSIS module as a cSimpleModule. Following this approach several active entities, i.e., POSIX threads, are bundled in one single cSimpleModule. The OMNeT-specific `handleMessage()` function is then used to realize the NSIS module's specific logic. Conditions to synchronize thread executions are signaled by boolean variables instead of using additional messages. This approach proves especially advantageous regarding the smaller amount of necessary messages and objects, and, finally results in a smaller memory footprint.

3.2 Message Handling

The GIST and Protlib implementations both use so-called FastQueues in which messages are managed whereas OMNeT++ uses a Future Event Set (FES) where all unpro-

cessed messages are stored. As the FES must be used anyway in order to activate the handleMessage() functions, we decided to insert messages intended for a FastQueue directly into the FES. In the existing implementation, message reception is performed by a so-called dequeue_timedwait() function call which blocks in case the FastQueue is currently empty or until a specific timeout occurred, respectively. This, on the other hand, leads to different points in time between the function call and the function's return which is especially problematic for the handleMessage() function that is processed at exactly one point in time.

This issue could be resolved by code refactoring where separate functions were introduced for the handling of arriving messages and timeouts, respectively. For the existing implementation this is just another level of indirection, but by following this approach the simulation can call these new methods directly via its handleMessage() function without the need to make use of thread-specific behavior.

3.3 Class Hierarchy and Initialization of Modules

In order to integrate an implementation into OMNeT++ some requirements with regard to the class inheritance hierarchy must be fulfilled. Firstly, all modules must be subclassed by the cSimpleModule class. Furthermore, the simulation kernel expects the SimpleModules to be used with an empty constructor. This implies that initialization data cannot simply be passed to the constructor.

The requirement to integrate the existing code seamlessly leads to the constraint of preserving the names of functions and variables as much as possible. Therefore, the originating inheritance hierarchy of the existing implementation must be preserved, too. The initialization of modules entails some conceptual problems. For instance, OMNeT++-specific parameters are accessible only once the OMNeT++ modules are constructed, i.e., data is not accessible within the SimpleModule's constructor but instead only upon the invocation of the initialize() method. In order to make OMNeT's initialization data accessible by NSIS-ka's classes, we decided to allocate dedicated memory space that could be later initialized by the initialize() function.

3.4 Simulating Multiple Hosts

The simulation of multiple logical hosts requires each single host to be clearly identifiable in order to access and manipulate data of one particular host. As mentioned earlier each host consists of several SimpleModules which are encapsulated by a CompoundModule as depicted in Figure 1.

Some of the data provided by the existing implementation was declared in a global range like the queue manager. The simulation, on the other hand, necessitates this data to be replicated for each single host, meaning that each host must be identifiable by a unique NsisId. In this case the NsisId is used by the simulation environment beforehand to identify the particular host whose data is to be accessed. However, this ID cannot simply be used as a CompoundModules parameter as each of OMNeT's parameters—and therefore also the NsisId—is only accessible once the initialize() method is accessible but not within the CompoundModule's constructor itself.

We decided to utilize OMNeT's way of module initialization where a ModuleId is provided according to the order in which all modules are initialized (following a breadth-first

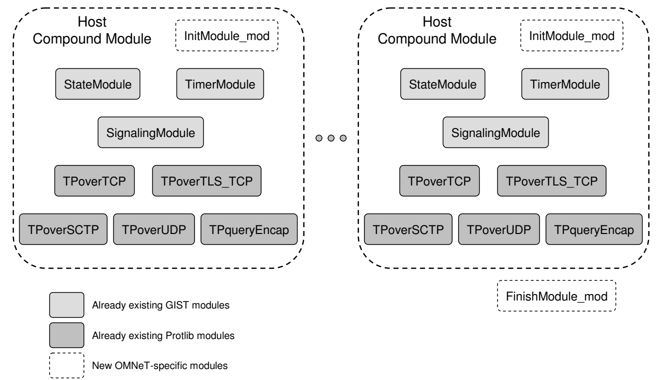

Figure 1: Modules being used by the NSIS-ka implementation and necessary module extensions for an integration into OMNeT++

search in the module's hierarchy tree). We introduced a dedicated OMNeT++ module (called init_module) which must be present at the very beginning of each CompoundModule within the NED file in order to be loaded prior to all other modules. A ModuleManager acts like a singleton for the entire simulation, increments an internal counter, and assigns the host's specific NsisId to the corresponding CompoundModule. This approach allows then for a seamless integration of the existing implementation into OMNeT++ as none of the existing public interfaces had to be changed.

4. EVALUATION

The resulting integration of the NSIS-ka implementation into OMNeT++ was evaluated by different test cases. We simulated a varying number of hosts with either one or two GIST connections per host. In each scenario a GIST connection is established by GIST's three-way handshake (Query, Response, and Confirm) and then used for ten subsequent Data messages. After that, periodic refresh messages are exchanged along the data path before a connection is torn down upon the expiration of a soft state timer.

The tests were performed on an Intel Core2 Duo P8700 CPU with 2.53GHz and 3GB of RAM, running in 32-bit mode on a Linux Kernel 2.6.31 with GCC 4.4.1 and libc6 version 2.10.1. OMNeT++ was used in version 4.0 in Cmd-Env mode and OppBSD in a developer version of the upcoming 4.0 release. The debug logging output was disabled in the NSIS-ka software.

4.1 Memory Usage

Figure 2 depicts the memory usage for a varying number of hosts as measured by the ps utility. The data reflects the value of the *Data Resident Set Size* (DRS), i.e., the data segment of the running simulation.

We performed measurements for a *Send_Direct* primitive, where messages are directly exchanged between peers without any lower layer interaction, and by using OppBSD with a complete TCP/IP stack underneath. Using OppBSD promises for more realistic simulation results and proves especially advantageous for obtaining tcpdump pcap files that can be used for an offline analysis afterwards.

We observe a higher memory footprint by using OppBSD which is a direct consequence of the socket memory buffers that are allocated by FreeBSD's TCP/IP stack. However,

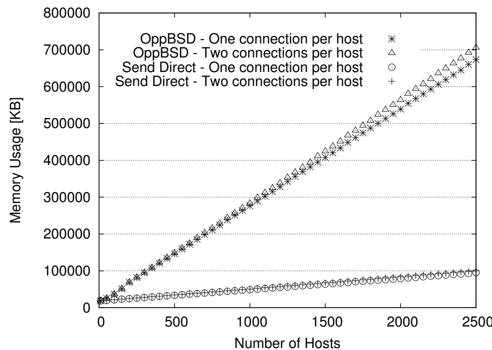

Figure 2: Evaluation of memory consumption

we obtain a *linear* growth rate showing that the memory consumption per host is constant. Table 2 summarizes some measurement values for a specific number of hosts.

# Hosts	Using OppBSD One Conn.	Using OppBSD Two Conn.	Using Send_Direct One Conn.	Using Send_Direct Two Conn.
10	18 825	18 961	18 634	18 638
100	36 653	36 801	21 366	21 634
1 000	276 485	283 077	48 702	50 646
10 000	751 853	758 469	324 194	345 994

Table 2: Memory consumption for simulations with a varying number of hosts in KB

4.2 Runtime Performance

Furthermore, we evaluated the simulation's runtime performance for a varying number of hosts. We used setups with hosts participating in one or two connections and by either using the *Send_Direct* primitive or by using the OppBSD TCP/IP stack, respectively.

In Figure 3 the resulting performance evaluation is shown for a varying number of hosts, ranging from ten simulated hosts up to 2 500 simulated hosts. We simulated 130 seconds of interaction and measured the totally elapsed time of the simulation's run.

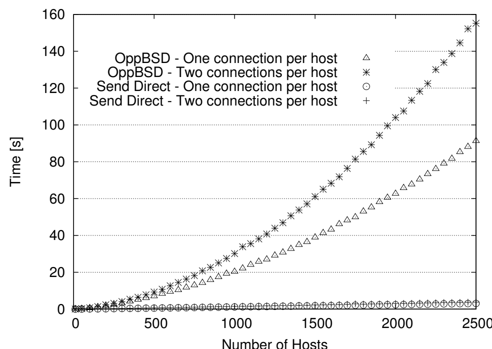

Figure 3: Evaluation of runtime performance

Without using the OppBSD TCP/IP stack underneath we can simulate a 130 seconds run with far more than 10 000 hosts, whereas 130 seconds are already simulated in *realtime* by simulating 3 500 hosts with one connection and by using the OppBSD TCP/IP stack (see Table 3).

# Hosts	Using OppBSD One Conn.	Using OppBSD Two Conn.	Using Send_Direct One Conn.	Using Send_Direct Two Conn.
10	0.030	0.037	0.012	0.011
100	0.572	0.703	0.103	0.121
1 000	20.450	30.168	1.164	1.482
3 000	125.684	221.646	3.723	4.600

Table 3: Performance evaluation of 130 simulation seconds with a varying number of hosts in seconds

5. CONCLUSIONS

In this work we presented the integration of an existing implementation of a protocol framework into OMNeT++. We discussed emerging challenges and outlined different design decisions to resolve these issues. By following this approach the NSIS-ka's GIST implementation can now also be used with an underlying OppBSD TCP/IP stack yielding more realistic simulations. Evaluations of the memory footprint and runtime performance show the feasibility of the chosen approach. Future work would comprise the integration of the existing code of the NSIS-ka's NSLP protocols, in order to perform comprehensive evaluations for Quality-of-Service signaling, i.e., QoS NSLP, in a large scale, too.

6. ACKNOWLEDGMENTS

We thank Stefan Hartte for his valuable help in elaborating the concepts of this work and his major contributions to the resulting implementation.

7. REFERENCES

[1] X. Fu, H. Schulzrinne, A. Bader, D. Hogrefe, C. Kappler, G. Karagiannis, H. Tschofenig, and S. V. den Bosch. NSIS: A New Extensible IP Signaling Protocol Suite. *Communications Magazine, IEEE*, 43(10):133–141, October 2005.

[2] Institute of Telematics. NSIS-ka – A free C++ implementation of NSIS protocols, February 2010. `http://nsis-ka.org/`.

[3] Institute of Telematics. OppBSD – A FreeBSD Network Stack integrated into OMNeT++, February 2010. `https://projekte.tm.uka.de/trac/OppBSD/`.

[4] C. P. Mayer and T. Gamer. Integrating real world applications into OMNeT++. Telematics Technical Report TM-2008-2, Institute of Telematics, Universität Karlsruhe (TH), February 2008.

[5] R. Bless and M. Doll. Integration of the FreeBSD TCP/IP-stack into the discrete event simulator OMNet++. In *WSC '04: Proceedings of the 36th conference on Winter simulation*, pages 1556–1561. Winter Simulation Conference, December 2004.

[6] H. Schulzrinne and R. Hancock. GIST: General Internet Signalling Transport. `http://tools.ietf.org/id/draft-ietf-nsis-ntlp`, June 2009. Internet Draft draft-ietf-nsis-ntlp-20.

[7] A. Varga and R. Hornig. An Overview of the OMNeT++ Simulation Environment. In *Simutools '08*, pages 1–10, ICST, Brussels, Belgium, 2008.