Universität Karlsruhe (TH)
Institut für Telematik

TELEMATICS TECHNICAL REPORTS

Integrating real world applications into OMNeT++

Christoph P. Mayer, Thomas Gamer
{mayer,gamer}@tm.uka.de

February, 27th 2008

TM-2008-2

ISSN 1613-849X
http://doc.tm.uka.de/tr/

Institute of Telematics, University of Karlsruhe
Zirkel 2, D-76128 Karlsruhe, Germany

Integrating real world applications into OMNeT++

Christoph P. Mayer
Institut für Telematik
Universität Karlsruhe (TH)
Germany
mayer@tm.uka.de

Thomas Gamer
Institut für Telematik
Universität Karlsruhe (TH)
Germany
gamer@tm.uka.de

ABSTRACT

The discrete event simulator OMNeT++ is nowadays used for network simulations in the majority of cases. Unfortunately, it is not possible to easily integrate real world networking applications into simulation models. This, however, would enable less complex and more efficient development and evaluation of real applications, especially of those that work in a distributed manner, in comparison to an evaluation in real world networks. We therefore present in this paper approaches to overcome the shortcoming of real world application simulation in OMNeT++ and discuss problems and solutions that arise in this context. Our preferred approach for integrating real applications into simulation models is based on an encapsulation of real applications as shared libraries that can be dynamically loaded by OMNeT++ at runtime.

Categories and Subject Descriptors

I.6.3 [**Simulation and Modeling**]: Applications

General Terms

Application simulation

Keywords

OMNeT++ integration, hardware-in-the-loop, real world simulation, application model

1. INTRODUCTION

When it comes to network simulation researchers often rely on the OMNeT++ simulation environment [13]. Due to its powerful discrete event simulation engine and lots of available network protocols it is very easy to simulate and evaluate networking protocols and applications using e. g. arbitrarily large and complex network topologies [5].

The integration of existing real world applications into simulators like OMNeT++ notably simplifies evaluation and further development of these applications since topologies, traffic patterns, and other parameters can be changed easily within a simulator. This is even more important in case of distributed applications, e. g. distributed anomaly-based attack detection [6]. The detection systems in this example are sparsely distributed on nodes within the network. Specific information, e. g. statistical traffic distributions which are locally measured, is communicated between various instances in order to improve detection of adverse events like distributed denial-of-service attacks or worm propagations. Thus, such attacks can be detected efficiently and as early as possible. The behavior of such an application in large networks and under varying conditions may be more easily evaluated using a network simulator than a real network. Thus, an easy integration of real world applications into simulators should be possible without the need for redesigning or re-implementing them for the OMNeT++ environment.

If networking protocols are implemented for usage with OMNeT++ protocol logic typically is integrated into so called *simple modules* using C++ code. By combining multiple simple modules into a *compound module* it is possible to implement more complex application logic, too. Thus, additional client and server applications can be integrated into the simulation model, e. g. in order to generate realistic background traffic. Integrating already existing real world applications into OMNeT++ simulations, however, is not as easy as implementing new applications specifically for OMNeT++. Problems arise, for example, if threads are used in real world applications. In addition to the complexity of actually integrating real world applications, serious simulation runtime problems may arise. These could result in time distortion, CPU scheduling issues, and process or symbol space problems.

Previous work mainly focused on the integration of real world TCP/IP network stacks into OMNeT++ or other network simulators [2, 8]. The overlay network simulation framework OverSim [1] uses socket connections to connect external applications to the simulation model. We are, however, not aware of further work that details challenges and solutions of easily integrating existing real world applications into OMNeT++.

The rest of the paper is organized as follows: Section 2 further motivates the need for integrating real world applications into OMNeT++. Different approaches for an integration of such applications – socket connection, source code integration, and shared library integration – are discussed and compared in section 3. We consider usage of shared libraries the best method for integrating real world applications into a simulation model. Therefore, this method is presented in more detail in section 4. Additionally, necessary adjustments that have to be made on applications as well as OMNeT++ are explained in detail. Section 5, then, details the problems that arise at compile time and runtime and describes solutions. Finally, section 6 gives a short summary and an outlook on future work.

2. WHY INTEGRATE?

Evaluating real world applications – especially distributed applications – based on real networks is a challenging task. Keeping the previously described example of distributed anomaly-based attack detection in mind, a real network must achieve several requirements in order to be usable for evaluation:

1. A large network is necessary in order to get meaningful evaluation results.
2. Control over all nodes in the network is needed, e. g. to deploy the attack detection system or to execute a distributed denial-of-service attack.
3. A realistic network environment including e. g. a realistic topology and realistic background traffic is needed.

These requirements apply to most distributed applications. Creating a large testbed for evaluation, however, results in high efforts for deploying necessary applications – attack detection as well as attack execution in our example – on different hosts in the network. Furthermore, building up such a large network – maybe consisting of hundreds or thousands of nodes – involves high costs for hardware. The need for different network topologies and varying network conditions additionally increases the expenses and administrative efforts. Furthermore, clients – and possibly human users – and servers have to be deployed in the network in order to create realistic background traffic.

Another possibility is to use an existing large network for evaluation of real world applications, e. g. PlanetLab [9] which provides control over the nodes. This ensure that no additional hardware has to be deployed and thus, reduces costs significantly. If such a network, however, is used it must be ensured that the evaluation does not disrupt normal network operation and that failure or malfunction of the network does not bias evaluation results. Therefore, such an operational network can not be used in case of an attack detection system since attacks would disrupt normal network operation but must be generated explicitly in order to get meaningful evaluation results.

All of the mentioned requirements can be easily fulfilled when deploying existing real world applications in a simulation environment like OMNeT++. Creating arbitrarily large network topologies is provided by topology generators like ReaSE [5]. They furthermore allow creation of varying network topologies. Control over the network is always given when running simulations, thus the second requirement does not impose a problem, too. In order to achieve a realistic environment self-similar network traffic [16] must be generated. Additionally, malicious nodes have to be integrated into the simulaton, e. g. based on the Tribe Flood Network [4] DDoS tool as shown in [5], which execute DDoS attacks.

3. INTEGRATION METHODS

There are different approaches that can be used in order to integrate real world applications into OMNeT++ simulation models. In this section we will describe three techniques for such an integration: socket connection, source code integration, and using shared libraries. Additionally, we will evaluate the complexity and applicability of each suggested approach.

The socket connection (see section 3.1) is e. g. used by the OverSim framework [1] in order to connect real world applications like SIP clients to the simulation model. Direct source code integration (see section 3.2) is the default way to integrate protocol and application logic into OMNeT++ [15]. Using dynamically loadable shared libraries (see section 3.3) to outsource application code is partially documented in [14, 15].

3.1 Socket connection

Connecting a real world application to the simulation using a socket has been implemented e. g. by OverSim and is documented in the OMNeT++ sockets sample [13]. It requires a simple module acting as proxy within OMNeT++. This proxy maintains a socket connection with the real world application. This method often does not imply any source code changes on the application and thus, is very easy to implement.

Socket connections can be used if an application has no need for lower layer protocols but just needs data from application layer. Anyway it is also possible to tunnel the whole network packet including all lower layers over the socket. In most cases this requires source code adaptations on the application to support tunneling of network packets. This means that it is not possible to, e. g., transparently connect an intrusion detection system (IDS) that uses a Libpcap [7] interface for packet capture to an OMNeT++ simulation using a socket connection.

Despite the fact that the integration using a socket connection is quite easy to perform, it suffers from the following problems:

1. CPU scheduling issues
2. Synchronization issues

Since an OMNeT++ simulation typically runs as fast as possible it consumes as much CPU time as possible. This results in the first problem: The external application may not get enough CPU time for its own operations and therefore, does not run smoothly (1). In order to avoid this issue higher process priorities may be assigned to the real world application. This, however, may result in problems for OMNeT++ if traffic sent by the application can not be processed fast enough and thus, causes the socket connection to break.

OMNeT++, on the one hand, performs a time discrete simulation in its own time domain: the *simulation time*. An application that is connected to a simulation using a socket connection, on the other hand, runs in its own time domain: the *wall clock time* which runs in real time. This causes time distortion between simulation and real world application that leads to inaccurate or even false simulation results (2). The solution that is implemented by OMNeT++ and OverSim is to use a special simulation scheduler that

slows down the simulation to real time. This causes both the simulation and the external application to run in the same time domain, i. e. the *wall clock time*.

This approach, however, results in further problems that may bias the simulation results. First of all, the approach to slow down the simulation is only possible if the simulation is able to run faster than real time. Depending on the number of modules used in the simulation and on generated traffic between the hosts in the simulated network this requirement must not taken for granted in all cases. Secondly, the external application is still not running in a time discrete manner and therefore, may not be synchronized with the simulation properly. The last drawback of slowing down the simulation is the extended simulation time: Executing a simulation that normally runs ten times faster than real time and provides ten hours of simulation time will end up running for ten hours real time instead of just one if socket connections are used for the integration.

3.2 Source code integration

Integrating source code directly into a simulation is the default approach when evaluating protocols using OMNeT++. This involves writing simple modules in C++ and compiling them using the OMNeT++ build environment. This approach enables easy development of protocols and small applications. Since all code is implemented directly into OMNeT++ and scheduled by OMNeT++ no time distortion can occur.

The direct source code integration technique suffers from the following problems that make it difficult to integrate existing real world applications into OMNeT++:

1. Real world applications consist of multiple source files and contain dependencies that have to be integrated into the OMNeT++ build environment.
2. The build environment for the real world application has to be reconstructed in the OMNeT++ build environment. This includes compiler and linker flags.
3. External application dependencies have to be integrated into the OMNeT++ build environment.
4. Features like timers and threads that are used in real world applications cannot be seamlessly integrated into OMNeT++.

Problems (1) to (3) concern the software build environment: Reconstructing the application's build environment within the OMNeT++ build environment involves integrating all source files and applying all compile and link time switches that are needed by the real world application. This can result in incompatibilities and unstable code: Compile time switches for character set, exception handling and threading, for example, can result in incompatibilities, break the build or produce runtime failures. Adding external dependencies into the OMNeT++ build environment (3) can further complicate the build or even make it impossible due to link time incompatibilities between the external dependencies and OMNeT++.

As OMNeT++ pursues a discrete simulation model the use of threading (4) causes problems (see section 5.1). This is a general problem that is not only related to the approach of source code integration. Furthermore, timer mechanisms must be considered: Total timer functionality has to be emulated by using OMNeT++ *self messages* (see section 5.2).

3.3 Shared library integration

Another possibility to integrate applications into OMNeT++ is to include binary code. We call this method *shared library integration*. It is quite similar to the integration of source code and involves most of its integration steps. As huge advantage the Shared library integration avoids the most challenging problem: project build management.

Whereas the source code integration requires major adaptations to the OMNeT++ build environment the approach of using shared libraries keeps the build environment for the real world application and OMNeT++ separated in a sound way. Compile and link time switches as well as external dependencies of the application can be retained. This enables different compile and link time settings for OMNeT++ and the real world application. In the majority of cases a real world application uses compile and link settings that can not be changed without breaking the application. Thus the Shared library integration enables the application to preserve its specific settings and integrate into OMNeT++.

Changing the output type of an application to shared library involves only small adaptations to the application build environment that can be implemented e. g. using command line flags for the GNU make build system [12]. This avoids costly maintenance of multiple project build environments for the application. Only one application build environment exists that can be used to build both the native application and the OMNeT++ compatible shared library.

3.4 Conclusions

The solution using shared libraries includes all advantages of the direct source code integration but heavily simplifies the problems regarding management of both OMNeT++ and real application build environments. The solution based on socket connections, is the easiest one but shows synchronization problems that may bias simulation results and additionally, does only work if simulation speed is faster than real time. We hence consider the shared library approach the best choice for the integration of real world applications into OMNeT++. The rest of the paper will focus on this integration technique. Section 4 will explain in more detail how to integrate real world applications using shared libraries. Section 5 finally focuses on emerging problems and gives solutions.

4. INTEGRATION STEPS

This section focuses on the integration of real world applications using the shared library approach introduced in section 3.3. Section 4.1 will detail on the required adaptations on real world application and OMNeT++. The necessary OMNeT++ NED environment will be explained in section 4.2.

4.1 Preparing the application

The first step of preparing the real world application for integration into OMNeT++ involves creating a simple module using the base OMNeT++ class `cSimpleModule`. The functionality of the application's regular `main` method has to be encapsulated into this simple module. The actual code of the `main` method will be split into the simple modules constructor and destructor as well as `initialize` and `finish` methods. An initialization based on multiple stages can be implemented using the simple module's `numInitStages` method.

As next steo the network abstraction has to be built. Applications like intrusion detection systems or network analyzers parse all protocol layers of a packet. Most often such applications provide their own parsers to access protocol contents. These parse the binary network packet and represent the contents in a structured and easily accessible way. The intrusion detection systems Bro [11] and Snort [10], for example, employ such structured protocol parsers.

OMNeT++ uses protocol classes that are transmitted within the simulation model as objects in the simulation process space. The OMNeT++/INET protocol classes, however, are different from real binary network data. It is thus not possible to use the OMNeT++ protocol objects directly for protocol parsing based on RFC-conform protocol parsers. Figure 1 compares a binary IP packet format which is RFC-conform and a binary representation of an IP packet used by OMNeT++/INET. The binary representation for the OMNeT++/INET IP packet was derived from the `IPDatagram` message definition in the `IPDatagram.msg` file. It can be easily seen that the binary formats are different and that normal protocol parsers can not be used for both formats. Thus, it is not possible to transparently operate the original protocol parsers of the real world application.

Integrating a real world application including the application's original parsers is possible by using an abstraction layer for the network data. This involves converting the OMNeT++-specific protocol objects into structures that are used inside the application. There exist two different ways to achieve this:

1. Mapping the OMNeT++/INET protocol objects to application-internal protocol structures.

2. Serializing the OMNeT++/INET protocol objects to binary format and enable the application to perform the parsing.

Which approach is chosen depends on the fact whether the application provides inbuilt protocol parsers or not. If the encapsulation of the OMNeT++/INET protocol parsers can be mapped to application-internal protocol parsers it is possible to develop a conversion layer that converts the OMNeT++/INET protocol objects into application-specific protocol objects (1). The architecture of such an abstraction layer is shown in figure 2. Care must be taken because OMNeT++/INET defines several constants like ICMP type and code values that are not RFC-conform.

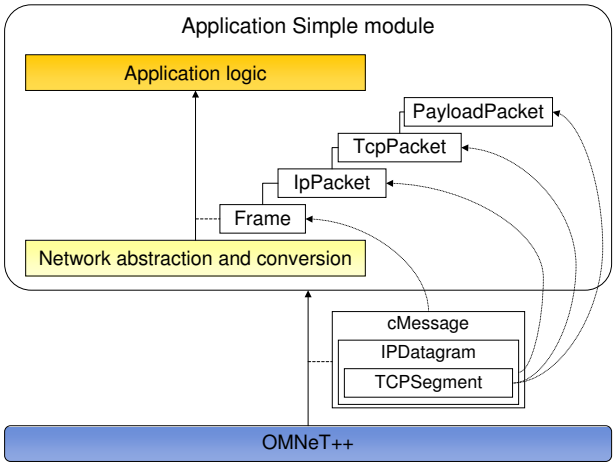

Figure 2: Abstraction layer for OMNeT++/INET protocol representation and application-specific protocol representation.

If the first approach can not be applied the serialization functions in OMNeT++/INET can be used to serialize the protocol objects into a binary Pcap [7] structure (2). This way a Pcap interface can be emulated that provides generic access to all network data including all protocol layers for the real world application. This approach can e. g. be used when the real world application does not employ protocol parsers but accesses protocol contents through the use of simple offsets into the binary packet data.

The packet delivery model in the application may differ from the event driven approach used in OMNeT++. In an OMNeT++ simulation packets are delivered through a callback function, i. e. in a push-based manner. Network access interfaces like Pcap allow pull-based mechanisms to be used instead. If the application uses a pull-based method to access network packets additional buffers must be implemented to emulate pull-based network packet access in OMNeT++.

Already integrated applications that need to communicate with other nodes in the simulation need adaptations, too. This is much easier since no protocol parsers or serializers have to be employed. To enable communication between nodes in the simulation an abstraction is needed that allows an application to transparently communicate with other nodes using e. g. sockets under a native environment and using the OMNeT++/INET facilities when running a simulation.

A sound object-oriented design of the real world application makes integration into OMNeT++ rather seamless. The modular architecture allows the exchange of e. g. network access layers for the OMNeT++ environment. This results in a high level of abstraction and enables the actual application functionality to run independently of the underlying network access methods. Badly designed applications may require more integration efforts but can be integrated into OMNeT++ nonetheless using the presented method.

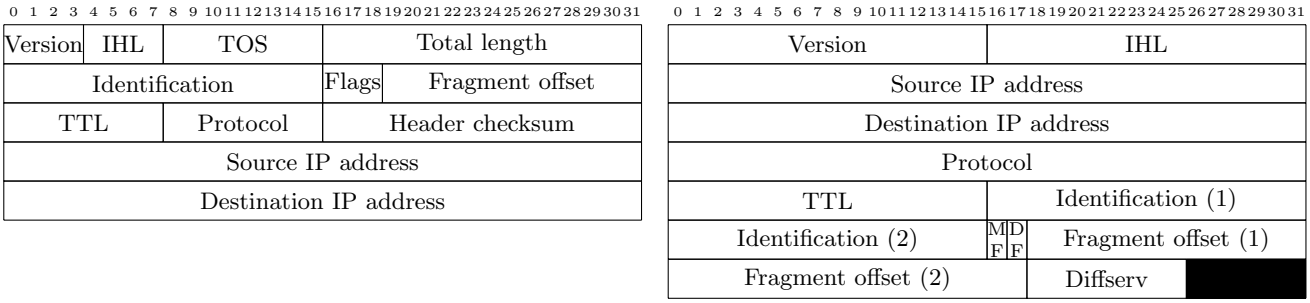

Figure 1: RFC-conform binary IP format (left) and OMNeT++/INET-based IP format (right)

Table 1: Flags for shared library building

Type	Flag	Meaning
Compiler	`-fPIC`	Generate position-independent code (PIC) that is needed for dynamic linking.
Linker	`-shared`	Produce a shared object that can be used to link a shared library.
Linker	`-rdynamic`	Instructs the linker to add all symbols to the dynamic symbol table. This is needed if the application dynamically loads further shared libraries which use functionality from the application shared library.
Linker	-o	Specify the output filename for the shared library, e. g. `libmyapp.so`

After adapting the source code using the simple module and building an appropriate network abstraction the application has to be built as a shared library. This applies only to the main executable component. Further shared libraries that are used by the application executable do not need to be changed. Changing the output type from executable to shared library needs adjustment of compiler and linker flags. Table 1 gives an overview of useful flags that may be necessary for compiling and linking a shared library with the GNU g++ compiler under Linux. Using a build system like GNU make [12] enables command line flags that can be used to select the runtime environment (e. g. native Linux or OMNeT++) and thus easily adapt the compiler and linker flags.

4.2 OMNeT++ NED environment

After the application has been prepared as shown in section 4.1 the OMNeT++ environment has to be built. The NED and configuration files for the OMNeT++ simulation are quite similar to the default approach.

First of all we adapt the `omnetpp.ini` file. It contains a `[General]` section which will be extended to command OMNeT++ load the application shared library that was built in section 4.1. In order to achieve this we add the following entry to the `omnetpp.ini` file:

```
[General]
...
load-libs="libmyapp"
...
```

This instructs OMNeT++ to dynamically load the file `libmyapp.so` at startup. This way the simple module that was defined in section 4.1 is now available for instantiation. It has to be ensured that the shared library `libmyapp.so` resides in a location where it can be found for dynamic loading.

The NED files that actually use the simple module look as if the source code was directly embedded, i. e. there is no difference to the default way of writing NED files.

5. CHALLENGES

The procedure described in section 4 enables the integrated application to startup and run within the OMNeT++ simulation. Nonetheless only simple applications will run without problems at this point. Most often problems will emerge that require special handling. These problems will be explained in the next sections and appropriate solutions are presented. The problems and solutions discussed in this sections also partly apply to source code integration (see section 3.2) and socket connection integration (see section 3.1) technique.

5.1 Threads

Threading is a direct contradiction to the discrete simulation approach followed by OMNeT++. A simulation performed by OMNeT++ is not multi-threaded and thus, implements no kind of concurrency. In real world applications threads are commonly used and receive great support by upcoming multicore processors. This results in a huge amount of software being written based on threads.

In case a multi-threaded application has been integrated as previously described one or both of the following problems may occur:

1. Application threads don't receive enough CPU time from the operating system scheduler and thus don't run smoothly.

2. Access violations occur when a thread tries to access the OMNeT++ simulation environment.

Threads that only perform little work and then finish are often not affected by too little CPU time. Such threads will run correctly and do not need any adaptation in regard to (1). Threads that e. g. wait for processing items of a queue possibly will run too slow since the OMNeT++ environment consumes all CPU resources. To resolve problem (1) threads have to be discretized and thus source code changes need to be applied. The following example will make this more clear: Consider a thread that waits for items in a queue to appear. Then it performs some work on all items until the queue is emptied. Discretizing this thread can be done by periodically calling a method that processes all items in the queue. The calling of this method can be implemented in the OMNeT++ `handleMessage` method of the application's simple module. This way the thread itself is discretized and processing is triggered by OMNeT++ directly. As processing time in a simple module is not added to the simulation time no time distortion can occur.

Another way to solve the CPU scheduling problem (1) is to adjust thread priorities. In case of shared library integration OMNeT++ and all application instances are treated as a single process. This means that it is not possible in this case to adjust process priorities. It is therefore necessary to adjust thread priorities. Adapting thread priorities, however, needs deep knowledge of application code and behavior as well as interaction of threads. The actual thread priority adaptations have to be performed based on the application source code.

The OMNeT++ simulation environment is at any time aware of the currently active code. Therefore a context pointer is used that can be manipulated using the class `cContextSwitcher`. A thread in the real world application – which does not necessitate discretization and thus has not been discretized – that accesses OMNeT++ functionality, e. g. through the application's simple module object, is not marked as active code because it has not been scheduled by OMNeT++ but by its own thread. Therefore, problems occur that result in access violations (2) because OMNeT++ is not aware of the active code. To circumvent this problem the macros `Enter_Method` and `Enter_Method_Silent` can be used before accessing simulation objects like the applications simple module. They use the class `cContextSwitcher` mentioned before to adjust the context pointer. This way OMNeT++ is aware of the currently active code and can handle message ownership and event scheduling correctly.

5.2 Timing issues

Applications often depend on time-based information, e. g. for the following functionality:

1. Requests for the current time.
2. Request that the current thread should sleep for a defined time interval.
3. Use of timers that periodically execute a defined callback function.

Each of the three features rely on time-based information. Typically, applications, on the one hand, are executed in the wall clock time domain. An OMNeT++ simulation model, on the other hand, is executed in its own time domain: the simulation time. Hence, occurrence of any feature within an application needs adaptation to run correctly in the simulation time domain.

The simulation time can be requested by a simple module using the `simTime` method. This method returns the simulation time as floating point number with double precision in seconds. Using preprocessor macros the request for the current time (1) can be easily encapsulated and transparently used by the application:

```
#ifdef OMNET_SIMULATION
 #define CUR_TIME (static_cast<unsigned long>     \
                  (cSimpleModule::simTime()*      \
                  1000.0))
#else
 #define CUR_TIME (static_cast<unsigned long>     \
                  (((double)clock())/             \
                  ((double)CLOCKS_PER_SEC))*1000.0)
#endif
```

This way the application is able to perform time requests transparently within an OMNeT++ simulation as well as a normal runtime environment like Linux.

Sleeping the current thread (2) for a defined amount of time must be done in respect to the simulation time domain. As the simulation time is not increasing uniformly, sleeping can not be done using a multiplier in respect to the wall clock time domain. Thus, sleeping mechanisms must be implemented using OMNeT++ specific features. OMNeT++ provides the `wait` method that simulates speeping mechanisms using self-messages and can be used by a simple module. Like the request for the current time the sleeping mechanism can be encapsulated and made compile time-dependent. This results in transparent usage of sleeping mechanisms for the application.

Timers (3) are implemented in OMNeT++ using self messages. These are messages that are scheduled with a user defined delay and then sent to the module itself. Periodic timers can be implemented by rescheduling incoming self messages. The self message functionality can be used to implement a timer management for the application that can handle multiple timers transparently. The timer management thus is able to handle multiple timers using different identifiers in the OMNeT++ self message.

5.3 Shared libraries and symbol space

As soon as the application's shared library is dynamically loaded by OMNeT++ the runtime environment resolves all pending symbols. This is performed on a process space-wide scope. Problems can occur if the application that was loaded using the shared library in turn loads additional shared libraries (e. g. plugins). If plugins rely on functionality of the base application symbol resolution errors can occur. The reason for this is that when OMNeT++ loads the application's shared library only symbols are mapped into the sym-

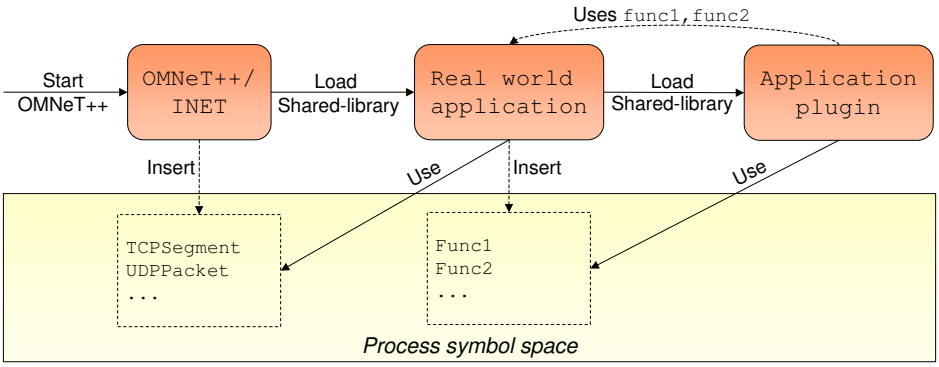

Figure 3: Shared library loading by OMNeT++ and by the integrated application.

bol space that are currently needed and thus, unresolved. Symbols that are not needed at this moment are discarded and not mapped into the symbol space. A plugin that may get dynamically loaded by the application at a later time may rely on this functionality. Because the functionality has not been mapped into the symbol space at startup loading the plugin will fail.

Figure 3 shows an exemplary scenario where OMNeT++ dynamically loads the real world application's shared library. All symbols that the real world application depends on, e. g. `TCPSegment`, are resolved and thus, can be used by the real world application. All symbols in the real world application that are currently needed are resolved, too. All symbols that are currently not needed are not inserted into the symbol cache. This results in `func1` and `func2` not being inserted into the symbol cache. If the application loads a plugin that relies on `func1` or `func2` this functionality – which resides in the application – is not accessibly by the plugin.

The solution to this problem is quite easy but requires a change in the OMNeT++ code itself. The OMNeT++ code for loading shared libraries is located in the file `loadlib.h` at `include/platdep/`. The method `opp_loadlibrary` uses the `dlopen` method to load shared libraries dynamically at runtime. This method call needs to get extended using the `RTLD_GLOBAL` option. Thus, the resulting function call looks as follows:

```
dlopen (libfname.c_str(), RTLD_NOW | RTLD_GLOBAL)
```

This causes all symbols in the application shared library to be mapped into the symbol space when loaded by OMNeT++. This includes symbols that are currently not needed. Thus, symbols that are needed by plugins can be resolved successfully, even at a later point of time.

Based on the changes described above all symbols in our example that are available in the real world application are inserted into the symbol cache and available for later usage by plugins. This means that `func1` and `func2` are inserted into the symbol cache at the time the shared library of the real world application is loaded. Thus, they are available during runtime, e. g., when the application loads a plugin that depends on `func1` or `func2`.

The nonexistence of C++ namespaces in OMNeT++ can cause further problems that need to be considered. Every class in the real world application is compiled into a symbol. This symbol is mapped into the process space and represents the entry point for the usage of the actual functionality. Two classes having the same name compete against the symbol space entry and will swap one another. This results in the wrong class being instantiated and actually used at runtime. Function calls on the class will result in the wrong memory location being executed and fail with access violation errors. The problem can be resolved by using C++ namespaces in the real world application or taking care that no class name in OMNeT++ and the real world application clash.

5.4 Process space issues

When deploying several instances of the same application on the same machine using e. g. Linux, every instance resides in its own process space. Under an OMNeT++ simulation multiple instantiations of the same application's simple module result in the fact that all instances reside in the same process space. This means that using the real world application on several nodes in the simulation network results in all application instances sharing the same process space. Every static variable or object is thus shared between all instances. Therefore, special care is needed if static variables and static objects like singletons are used.

The shared process space, however, can also have positive effects on the simulation performance: Shared memory pools, e. g. by using the Boost Pool Library [3], or the usage of singleton objects result in smaller overall memory usage and less CPU time consumption. This may speed up simulation execution.

6. CONCLUSIONS

Integrating real world applications into a simulation environment like OMNeT++ enables an evaluation of their behavior in large simulated networks. Furthermore, it is possible to design and evaluate the behavior of distributed real world

Table 2: Advantages and disadvantages of discussed integration techniques.

Integration technique	Advantages	Disadvantages
Socket connection	Applications like servers and clients can be integrated without changes to the application source code.	CPU and timing issues make this integration technique unstable, inefficient or even impossible to perform.
Source code integration	No CPU or time domain issues, thus no bias. Straightforward integration method. Easy to integrate small and simple applications.	Need to reconstruct application build environment with OMNeT++. Applications with special compiler and linker settings can not be integrated due to incompatibilities with OMNeT++. Source code adaptations needed.
Shared library integration	All advantages of source code integration. Avoids problems of build environment and incompatibilities.	Source code adaptations needed.

applications more easily and cost-efficient than in real networks. We presented several techniques that can be used to integrate existing real world applications into OMNeT++ and discussed their pros and cons. Focusing on the approach to use shared libraries for integration we presented detailed instructions how to realize this solution. Additionally, emerging challenges are explained and solutions are provided. Table 2 gives a concluding overview of advantages and disadvantages of the different integration techniques discussed in this paper.

A well designed network application can be integrated into OMNeT++ more easily than badly structured code. Anyway the integration can be time consuming and complex. It would be desirable to have better support by OMNeT++ for the integration of real world applications. Future work should focus on OMNeT++ extensions that can easily deploy existing real world applications without the need for major adaptations on the application side.

7. REFERENCES

[1] I. Baumgart, B. Heep, and S. Krause. OverSim: A Flexible Overlay Network Simulation Framework. In *Proceedings of 10th IEEE Global Internet Symposium (GI)*, pages 79–84, May 2007.

[2] R. Bless and M. Doll. Integration of the FreeBSD TCP/IP Stack into the discrete event simulator OMNeT++. In *Proceedings of the 2004 Winter Simulation Conference*, pages 1556–1561, December 2004.

[3] S. Cleary. Boost Pool Library. http://www.boost.org/libs/pool, 2000.

[4] D. Dittrich. The "Tribe Flood Network" distributed denial of service attack tool. http://staff.washington.edu/dittrich/misc/tfn.analysis, October 1999.

[5] T. Gamer and M. Scharf. Realistic Simulation Environments for IP-based Networks. In *Proceedings of the OMNeT++ Workshop*, Marseille, France, March 2008.

[6] T. Gamer, M. Scharf, and M. Schöller. Collaborative Anomaly-based Attack Detection. In *Proceedings of IWSOS*, pages 280–287. Springer, August 2007.

[7] V. Jacobson, C. Leres, and S. McCanne. Tcpdump Pcap. http://www.tcpdump.org, 2000.

[8] S. Jansen and A. McGregor. Simulation with real world network stacks. In *Proceedings of the 2005 Winter Simulation Conference*, pages 2454–2463, December 2005.

[9] PlanetLab. PlanetLab: An open platform for developing, deploying, and accessing planetary-scale services. http://www.planet-lab.org, 2002.

[10] M. Roesch. Snort. http://www.snort.org, 2001.

[11] R. Sommer. Bro: An Open Source Network Intrusion Detection System. In *Proceedings of the 17. DFN-Arbeitstagung über Kommunikationsnetze*, pages 273–288, June 2003.

[12] R. M. Stallman and R. McGrath. GNU make. http://www.gnu.org/software/make, 1988.

[13] A. Varga. The OMNeT++ Discrete Event Simulation System. In *Proceedings of the European Simulation Multiconference*, pages 319–324, June 2001.

[14] A. Varga. How to extend INET with your own C++ code. http://www.omnetpp.org, June 2005.

[15] A. Varga. *OMNeT++ Discrete Event Simulation System User Manual*, 2005. Version 3.2.

[16] W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson. Self-similarity through high-variability: statistical analysis of ethernet LAN traffic at the source level. In *Proceedings of ACM SIGCOMM*, pages 100–113, February 1995.