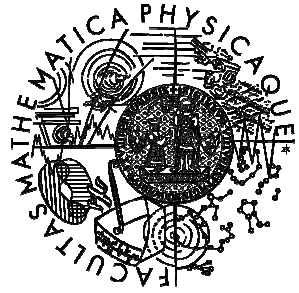
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Pavel Novák

Simulation of Network Structures

Department of Software Engineering
Supervisor: RNDr. Ing. Jiří Peterka
External Advisor: Mgr. Petr Votava
Study Program: Computer Science, Software Systems

I would like to thank Jiří Peterka for his encouragements to this text and Petr Votava for his suggestions to the simulator design. Special thanks to Vendulka and my parents for their support throughout my studies.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with lending of this work.

9th August 2006, Prague Pavel Novák

Table of Contents

Title: *Simulation of Network Structures*

Author: *Pavel Novák*

Department: *Department of Software Engineering*

Supervisor: *RNDr. Ing. Jiří Peterka*

Supervisor's e-mail address: *jiri.peterka@mff.cuni.cz*

Abstract:

The simulation of network structures can be an effective method for example in teaching, research, or testing the network software, in order to lower the cost of building the real network structure that would be otherwise needed. Moreover, the simulated network can provide some advantages, e.g. simplified control and monitoring, statistical data collecting or visualization of the network behavior.

This work focuses on the usage for teaching and testing; illustrates that existing network simulators are not always suitable for this purpose, suggests the architecture and design of a new solution, and offers implementation of the proposed simulation tool.

Keywords: simulator, computer network, teaching, testing, NetSim

Název práce: *Simulace síťové struktury*

Autor: *Pavel Novák*

Katedra (ústav): *Katedra softwarového inženýrství*

Vedoucí diplomové práce: *RNDr. Ing. Jiří Peterka*

E-mail vedoucího: *jiri.peterka@mff.cuni.cz*

Abstrakt:

Simulace síťové struktury může být efektivní metodou použitelnou např. při výuce, výzkumu nebo testování síťových monitorovacích programů, všude, kde by jinak byla potřeba mnohem dražší výstavba reálné sítě. Navíc přináší simulace sítě některé výhody, jako je zjednodušené ovládání simulované struktury a jejího sledování či sběru statistických dat, nebo i vizuální znázornění sítě a jejího chování v průběhu simulace.

Tato práce se zaměřuje na použití pro výuku a testování, ukazuje, že stávající dostupné simulátory často nejsou nejvhodnější volbou pro toto použití. Navrhuje strukturu nového simulátoru pro daný účel, součástí práce je i jeho implementace.

Klíčová slova: simulátor, počítačová síť, výuka, testování, NetSim

1. Introduction

The task of network simulation has been solved for a long time and many simulators have been created. However, a lot of them were written for special purposes – for testing just one network component or protocol. It is obvious that those simulators cannot be widely used and this work does not not deal with them. On the other hand, the goal of other simulators is to be extensible and to allow others to add their own modules for currently unsupported or new protocols, create new network devices etc. This is the kind of simulators the thesis deals with. The most widely known simulator of this kind is probably NS[1].

Let's take a look at reasons why network simulators are needed or, at least, why they are worthwhile and helpful. Here are some scenarios where the network simulator is benefical:

1. Education

   Students can build a network and see how it works without the need of hardware; try network tools or develop network components.

2. Testing

   Simulators can help test network management tools or other network applications, which can be deployed at a complex virtual network structure.

3. Demonstration

   Network application or management tools can be demonstrated to potential customers without affecting their real network structure.

4. Security

   Running a virtual network within the real network can lower the chance that an attacker will compromise vital real system, since he/she will get confused by other virtual nodes, and the network administrator has a chance to notice an illegal activity before the real system is compromised [8].

5. Designing new network protocols

   Protocol designers need to test their ideas in real environment to see if they are behaving in the way they expect. Building the real network structure is very expensive and these days, when computer networks are quite large, testing in such environment is inevitable.

As summarized in Appendix A, many general purpose simulators have been developed; also many of them are well-designed, so they are extensible and allow adding new features or communication protocols with quite a little effort. However, the following discussion will show that there are some reasons to develop a new one.

[1] Currently NS-2; for more information and the link see Appendix A.

Requirements Analysis

The original thesis submission states:

The task is to simulate a network structure for use in teaching, testing applications for network monitoring, or creating "hacker-traps" (so-called honeypots). The author should become familiar with network simulation, design and implement either his own solution or use existing frameworks (e.g. open source framework Honeyd). The solution should simulate switches, routers, and host computers so they would behave like real and react at least to the basic network protocols like ICMP and SNMP.

However, as will be explained, the final usage scenarios proposed here have different requirements and the development of a system that meets all of them is not viable.

The use for teaching

Various uses of a simulator for teaching are possible. Firstly, it can help the lecturer to show a network behavior for some cases, for example network reconfiguration after a failure of some network component, or how routing information is propagated through the network. The simulator used for this purpose should be able to visualize the network structure, network data transfer, pause the simulation. Naturally, it should implement network features that would be taught. Moreover, it would be better to simulate the network at the link layer, not only network layer, and packets being transferred through the network should be displayed in the same format as the real packets.

Secondly, students can use the simulator themselves to either build their own network and see how it works (in this case the requirements coincide with the use by the lecturer; moreover, the simulator control and creation of the virtual network structure should be quite simple), or, they can implement their own network components, such as a modules for routing, switching, etc. This requires the simulator to be well extensible and to provide a simple interface to add new components.

Thirdly, a simulated network can be used to teach network administration and administration tools; nevertheless, this is almost the same case as application testing.

Testing (and demonstration of) applications for network monitoring

This usage has a bit different requirements than teaching. Firstly, a virtual network setup does not have to be necessarily so intuitive, since it will be used mainly by specialists in computer networks. However, it would be a considerable advantage if the intuitiveness would be present along with other features.

The key requirement is the support for all network technologies and protocols that are mainly used by network monitoring programs. It includes primarily ICMP, SMTP (preferably all versions) and protocols for dynamic routing (OSPF, RIP, perhaps BGP). The second requirement, that will be necessary in some cases, is the possibility to simulate large networks, hundreds (thousands) of nodes. Creation of such network manually would be a painful work; having some tool for automatic network generation would be handy.

Honeypots (honeynets)

Honeynets is a pseudonym for computer networks (real or virtual) that are deployed only for monitoring attackers activity, warn, and provide information about the attack scenario to help prevent attackers from being successful in their activity[2].

Therefore, the main requirement for the network simulator acting as honeynet is to have support for distinguishing attacks from ordinary network traffic[3] and to log the operations performed by an attacker; possibly warn a network administrator that the attacker is trying to compromise his/her network. Wide range of network protocols does not have to be necessarily implemented; no graphical interface is necessary. However, a possibility to add new features to be simulated should be a priority, since it will allow adding new simulated items as attackers change their targets to correspond with known vulnerabilities of real systems.

The following table summarizes the results of a previous discussion.

Requirement	*Teaching*	*Testing*	*Honeynets*
Simulation at link layer level	✔	●	✘
Network and data flow visualization	✔	✘	✘
Easy extensibility	✔	●	✔
Wide range of protocols support	●	✔	✘
Large networks simulation	✘	●	✔
Writing logs for activity in the network	✘	●	✔

✔ required ● required in some cases ✘ generally not required

Many of the requirements presented in the table are not contradictory; however, some combinations would be hard to achieve. For example, link layer simulation needs more computation resources than only network layer, and therefore the maximum number of nodes will be much lower even if it would be efficiently implemented. The same applies for visualization; however, it could be a feature that does not have to be used throughout the simulation process.

Moreover, after a closer look at the table, it is evident that requirements for teaching and virtual honeynets differ. The use for testing is somewhere in between and the requirements for teaching and testing are not contradictory.

First Conclusion

Development of a simulator that meets all needs for the usage scenarios mentioned in the assignment would result in a complex solution, which would compel many trade-offs. Since

[2] A lot of information about honeynets can be found for example at http://www.honeynet.org

[3] Which is generally not so hard since any activity in a honeynet is usually mean; regular network traffic should not be targeted to the honeynet.

the requirements for honeynets differ from others and the author and advisor are much more interested in testing and teaching, this work will not be focused on honeynets.

From now on, the text is oriented only to teaching and network software testing purposes.

Existing Simulators

This chapter provides an overview of existing simulation tools, describes their usability and shows that there is a scope for another network simulator to be developed.

The simulators mentioned here are those that can be used for teaching and network monitoring tools testing. More simulators that the author found and that he considered to be somehow important are listed in Appendix A on page 54.

NS-2 [11]

This is probably the oldest widely used general-purpose simulator. It is written in C++ and many contributors have developed extensions for wide range of protocols. Its primary usage was intended for designing new protocols and testing their behavior.

Its use for teaching is a bit complicated. First, it supports only Unix-like operating systems; students using Windows are forced to learn much more to use it than only the program itself. One can object that the student who is learning computer networks should be familiar with Linux systems since they are more often used in networking than others are. However, as the NS installation is not straightforward and can make some problems to the student that is used to Linux too, this is not the right way how to begin learning Linux.

Moreover, the creation of the simulated network requires writing Tcl[4] scripts, which could be a serious problem for some students. There is a "network animator" called Nam, which can replay the previously executed simulation and – in the latest releases – build a network structure. However, it is completely bound to Linux and OTcl packages, which require compilation of the source code before installation.

In conclusion, NS-2 is a great tool for researchers and in some cases for testing, but it is too complex for non-experienced users. A useful tool for teaching should provide an easy installation (preferably only copying files) and a user-friendly (graphic) environment.

CNET [9]

Cnet was developed at The University of Western Australia and was intended primarily for teaching; students can write their own protocols. It requires Linux operating system; even though it is much more simple than NS-2, the installation also needs a source code to build it and some settings require a deep knowledge of the OS structure. Therefore, the preferred usage is probably the installation by an administrator in the lab where the students should work on their assignments.

[4] Tcl stands for "Tool Command Language"; more information at Tcl homepage: http://www.tcl.tk/

It cannot be used for testing purposes; firstly, it does not support connecting to the real network, secondly, no library with protocols implementation is available.

JNS [16]

Java Network Simulator is a Java version of NS-2, but obviously it had to be developed from scratch and almost all NS-2 features and libraries are missing. Instead of writing scripts in Tcl, they have to be written in Java. There is no graphic environment to build the network structure, nor can it be controlled interactively while the simulation is running. Since almost no libraries with protocols are present, it is unusable for testing. It can be used for teaching, but requires students having the knowledge of Java to be able to make their own simulations.

OPNET [12]

OPNET provides a set of tools for simulation that include a graphic environment for network modeling (with wireless mobility support), displaying statistics. Many pre-defined types of nodes are present and almost all widely used protocols and technologies are supported. However, the use for teaching is available only to universities and the conditions are strictly limited. The software can be installed in the university lab only, the license is only to six months and some reports about teaching experiences should be sent to the OPNET Company.

The supported operating system is Windows only. The source code for protocols implementation is available, the application source code is closed. The application is quite sophisticated; it provides many settings for almost everything and it may be difficult for the beginner to get into it. Therefore, it is intended rather for the networking professionals than students. Anyway, without those license restrictions, it would be a nice tool.

AdventNet [1]

AdventNet is another commercial simulator, its properties are almost the same as for the OPNET. Even if there are no remarks on licenses for teaching/education at their website, after a question they replied promptly and provided a 6-month full license.

On the other hand, it provides less flexibility than OPNET. Modification/addition of new network components is very limited, just a configuration file or setting MIB values is available. There is no support to add new protocols; therefore, students cannot implement their own solutions. Moreover, there is neither data visualization nor any statistics available; it is not a convenient tool for teaching. However, since network programs usually need SNMP which is supported quite well, it is the right application for testing and demonstration of network management tools (although it is a bit limited by no support for interconnection to the real network).

NCTUns [17]

NCTUns seems to be the most suitable simulator for the specified purposes. It includes a graphical network topology editor, new protocol modules can be added, the protocol stack of each node modified, and the virtual network can be interconnected to the real one. Moreover, many protocols are currently supported and the simulator writes a log file during the simulation, which can be later used for data flow visualization.

However, NCTUns has one main disadvantage: it actually requires a dedicated machine for practical usage. This is because Fedora Core 4 is the only supported system; the simulator is closely associated with the kernel and it uses kernel protocol stack for simulated nodes. (The kernel is patched during the installation and the simulation can be executed using the new version of kernel only.) Moreover, root privileges and no active firewall is needed for many operations. These demands prevent it from being installed in the computer lab or on any other computer that is in daily use for other tasks. Hence, it is impractical for teaching; however, having a dedicated machine is usually no problem while testing the network applications.

Other Simulators

QualNet [14]

QualNet is a proprietary network simulator. Although there is some version available for universities and the author of this thesis asked the Scalable Network Technologies for a copy, they did not reply in time so the comparison to this product is not available.

As can be read in the datasheet, graphical tools for network designing and analyzing are available; however, the proprietary license and closed source code make the use of the simulator and creation of the extensions difficult.

REAL [7]

REAL was developed at Cornell University for research of flow and congestion control. It is completely unusable for teaching since it does not run on the i386 platform that is in almost all computer labs and it is probably the computer students have at home. A port to i386 was created, but it works with FreeBSD 2.0.5 only. Similar arguments can be applied to the usability for testing.

Moreover, it seems that the simulator is not maintained anymore since the year of its last release is 1997.

Simulator Comparison

The previous paragraphs described the existing implementations of simulators; to be able to make some conclusion, the results should be compared. Recall that this work is concerned to the teaching of networking, network protocols, and to the testing of network monitoring applications. Obviously, this influences the requirements selected for the comparison.

The following table summarizes which requirements needed for the intended usage are met by the existing simulators. The numbers are references to the comments below the table.

Requirement	*NS-2*	*CNET*	*JNS*	*OPNET*	*AdventNet*	*NCTUns*
Easy installation	✗	✗	✓	✓	✓	✗
Requires writing no code to build simulated network	✗	✗	✗	✓	✓	✓
Simulation at link layer level	✗	✓	✗	✓	✗	✓
Network structure visualization	● [1]	✗	✗	✓	✓	✓
Data flow visualization	✓	✗	● [1]	✓	✗	✓
Extensibility of network components	✓	✗	● [2]	? [4]	✗ [3]	✗
Extensibility of supported protocols	✓	✓	● [2]	✓	✗	✓
Wide range of protocols supported	✓	✗	✗	✓	✓	✓
Interconnection to the real network	✓ [5]	✗	✗	? [4]	✗	✓
Runs on Linux	✓	✓	✓	✗	✓	✓ [6]
Runs on Windows	✗	✗	✓	✓	✓	✗
Source code available	✓	✓	✓	✗	✗	✓

✓ present ● not completely present

✗ not present ? not known

1) The visualization is available by Nam (or some other tools) that reads the dump created during the simulation process.
2) Since the whole source code is available and the simulator is designed to be extensible, new components can be created. However, there is no special support for that.
3) A new type of device can be created, but changes to its behavior are very limited; only its MIB values can be changed and IOS commands added.
4) The author was unable to find any information for the particular simulator.
5) This feature is available thanks to a third party add-in.
6) Only Fedora Core 4 is officially supported.

The ideal simulator for the target usage would have the ✓ sign in all rows. None of the examined real simulators fits those needs in all cases; the most suitable does not meet two requirements, the majority at least four requirements.

Second Conclusion

Most of the existing simulators are not very feasible for teaching computer networks and testing of network monitoring tools. The most suitable one, OPNET, is commercial; its university licensing is very restrictive, for example it does not allow students to install the simulator to their own computers, allows the installation at maximum 30 copies, the license should be renewed every 6 months and reports about teaching experiences should be written.

The second one that satisfies almost all requirements, NCTUns, has nearly all the features; however, it is bound to Fedora Core 4 kernel, which restricts its practical usability.

Therefore, the development of a new simulator that will extend the current set of simulators, which could be considered for the use for teaching and testing, is reasonable[5].

Project Goals Revisited

Let us go back to the requirements that were considered while comparing the existing simulators.

1) Easy installation
   That means preferably just copying files, or some simple installation wizard. Linux guru would appreciate a possibility to compile the application from source code (which might be also possible), but a regular user wants to put as little effort as possible. Therefore, binaries (along with source code) should be distributed.
2) No code when creating the simulated network
   A graphic tool for designing the virtual network would be the best; if not a graphic environment, some easy to understand configuration scripts should be available.
3) Link layer simulation
   Since the simulator will be used also for teaching, the simulated network should correspond to the networks in the real world. There are not only routers, but also switches and other components operating at link layer level.
4) Network structure and data flow visualization
   Network structure visualization is related to the creation of a network structure in a graphical environment. Data (packets) visualization might be needed for teaching while demonstrating network behavior and also for debugging purposes.
5) Extensibility
   Providing an easy way to add “third party” components to the simulated network increases the usability for both teaching and testing. Students can add their own protocol implementations, while developers of networking tools can add the currently unsupported functionality they need. Therefore, the addition of new components should not require simulator recompilation; just the modification of configuration files should be necessary.
6) Wide range of protocols
   This requirement does not affect design decisions. It is just about the amount of work that will be done. Hence, it is obvious that the first version of the simulator will include only a few protocols, preferably the TCP/IP suite.

[5] Moreover, the author is interested in computer networks and development of a network simulator from scratch would be a worthwile experience for him.

7) Multiplatform solution
   The simulator should be ported at least to two leading platforms: Windows and Linux.

Keeping these things in mind, the programming language and technology for development should be chosen.

First of all, it should be considered whether there is some platform for network simulation already developed. The use of honeyd, which is mentioned in the thesis submission, is impossible, since it does not support link layer simulation; moreover, it is intended mainly for honeypots. Another possibility would be to use JNS as a basis, but the implementation is very small; it would not save much time and would restrict further design decisions – the author could not find any other suitable framework that would meet the requirements described above. Therefore, he decided to implement a completely new simulator.

Multiplatform and extensible solution requirements support a decision for environments like Java or .NET. These also correspond with an easy installation, since some intermediate code that can be just copied can be usually distributed. Moreover, graphic applications created in Java can run on any computer where JRE is installed.

On the other hand, link layer simulation denotes that the simulator will require more computer resources than just the simulation on the network layer – native code would be faster than Java or .NET managed environment. However, this is the only reason to use traditional compilers and run native system code. The author's experiences with .NET show that it is not much slower than native code, since the intermediate code is recompiled to native before execution.

The managed environment will also simplify the development. The last choice between Java and .NET is needed. Here, the greatest factor is probably the author's experience in C# programming. Moreover, because, firstly, the .NET Framework for Linux is available and will be hopefully also implemented for other platforms in the future and, secondly, .NET will make the extending of the simulator even more simple since there are more programming languages the extender can select from[6], the last decision is .NET.

In conclusion, the project goals are summarized in the following statements that arise from the previous discussion:

- A managed environment should be used for development, .NET in particular.
- The simulator should be simply extensible allowing addition of new protocols, types of nodes and applications running at the virtual nodes. (The first implementation will consist of TCP/IP protocol suite emulation and some basic example applications.)
- A graphic interface simplifying the creation of virtual network structure should be provided.
- The user should not be forced to write any code if he/she uses only predefined (implemented) components.

[6] Not only C#, Visual Basic, J#, or C++, but also others made by third parties like PHP, Python, and other. A complete list can be found e.g. at http://www.gotdotnet.com/team/lang/

- Target platforms should be Windows and Linux[7].
- The simulation should be done at the link layer (Layer 2 in ISO/OSI model). Therefore, the simulation of network technologies has to be supported. (Ethernet, since it is the most wide-spread technology, will be provided as an example.)
- It should be possible to connect the virtual network to the real physical network.
- Nodes and links should be capable of shutting down/resuming to simulate network failures.

It was said that the best simulation behaves like a real network. Network traffic is generated by applications running on virtual nodes, so the applications are an important part of the simulator. It is not feasible to simulate all necessary applications; providing a way to run the existing applications on the virtual network nodes is an additional goal of the project.

7 Windows XP and Fedora Core 5 have been used for development and testing.

2. Simulator Architecture

The second chapter introduces the simulator architecture, explains the decisions made, and defines the naming convention for some network components used throughout this text. The results presented here will then be used in the following chapter elaborating the implementation details.

From here, the name *NetSim* will be used for references to the new simulator that has been developed.

Links and Nodes

The simulator should be designed to be well extensible; therefore, it should consist of several units with exactly defined interfaces, allowing the extender to write only one small piece of code and do not force him/her to understand the whole simulator structure. It is straightforward that those units will correspond to the components of a real network, provide basic network functionality such as transferring data from one node to another, switching and routing, etc. Nevertheless, for the simulation environment, some additional components might be needed; for example to launch applications or transfer data from a simulated network to the real network and vice versa.

There are two kinds of basic network components: cables (metal or optical, or one can imagine “virtual” cable in case of wireless connection) and some nodes that are interconnected with the cables. For the purpose of this document, they will be called *links* and *nodes*. However, there are many kinds of nodes and links in the real world and NetSim should be able to emulate them.

The simulation of links is much easier than for nodes. The purpose of the link is always the same: to transfer data from one place to (one or more) other places, with some constant or variable delay, and sometimes a loss or corruption of carried data. Compared to ISO/OSI network model, the links simulate physical and a part of link layer. Sometimes the situation is slightly more complicated; for example when simulating a ring-based network such as FDDI, it is not possible to just hand the data over to the link and not care about them any more – they will come back and should be removed from the link. In the case of Ethernet, the node should be informed about the collision and the need of retransmission. Therefore, the node might be required to participate in data transfer over the link.

In the case of nodes, the simulation is not so straightforward as there could be many kinds of nodes; just simple repeaters and switches, or complex machines running web and ftp servers, firewall, capable of routing etc. Because such nodes can be connected to various types of links, it will be effective to divide the link-specific part from the other node functionality. (Recall that for some link types, node cooperation is required.) Such parts of nodes are called *adapters*. Moreover, because the node can be connected to more than one link and link types can vary, each node can have more adapters. One can think of the adapters as of network cards with drivers.

The previous paragraphs described a data transfer between nodes. Nevertheless, what should the node do with the incoming data? They should be probably passed to some application running on the node. However, how should the node recognize which application it is? In the real networking world, incoming data packet traverses through the so called “protocol stack”, packet headers are checked at each stage, eventually removed, and the rest is passed to the higher level in the stack. This process should be emulated on each virtual node; each node can support various protocols that should cooperate with each other. Those protocols should be independent of applications running on the node – every application can use more than one protocol to communicate. So, what is the answer to the questions given at the beginning of the paragraph? The packet should be given to the first appropriate protocol implementation in the protocol stack. Such answer yields a problem: identifying an “appropriate” protocol. In general, there are two basic solutions: either the node will have some logic to recognize it or it can pass the packet to all protocols and they will decide themselves if it is “their” packet, which they should accept.

The first idea requires the node to know every protocol that might be possibly used, so that, with addition of a new protocol to the stack, the node code would have to be modified, which is inconsistent with the requirement of as easy as possible extensibility. From this point of view, the second solution is better – the implementer of the new protocol will make a logic recognizing new kind of packets. However, for correct packet processing the node cannot pass incoming packet to all possible protocol implementations, because there might be two of them accepting the packet and there would be two (different) reactions to the packet. Instead, the node will pass it to the protocol implementations in some sequential order and stop after the first that accepts the packet is found.

A strange phrase “protocol implementation” has been used several times in previous paragraphs. Let us define it as a *module* for now; the definition will be extended later.

In the real networking, a protocol stack is not only a sequence of modules; more than one module can be layered over another. For example, TCP and UDP protocols are layered over IP protocol. On the other hand, one module can be layered over more than one too: e.g. IP over Ethernet and serial line. However, that solution requires not only the node maintaining the list of modules, but also each module having to store its own list of the modules on the upper layer. Moreover, as described above, the packet will be passed to the modules in sequential order until it is recognized by some module. This approach can be applied not only when the packet is coming to the protocol stack, but at each layer. As a result, the whole protocol stack can be represented as a sequence of modules, if each module has its own logic to recognize whether it should process the incoming packet.

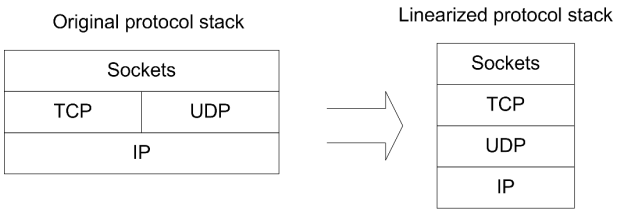


Figure 1: Protocol stack linearization

An example of such “linearization” is shown in Figure 1. If a packet is coming from the bottom, the first receiver is always IP. In the original protocol stack, IP module decides whether to pass it further to TCP or UDP. In the linearized stack, it passes the packet always to UDP, it decides whether it is an UDP packet; if not, it passes the packet up unchanged and TCP module has a possibility to process it if it is a TCP packet.

This linearization approach, where the module just takes a packet, processes it, and passes it up without knowledge of modules layered over has another advantage: other modules can be inserted into the stack without modifying the existing ones. For example modules for encryption or data counting.

In Figure 1, there is a “Sockets” module layered over TCP and UDP. It is clear that this module does not represent a protocol layer; however, it is useful to make it a part of protocol stack too; it allows for example data encryption between sockets and TCP/UDP. Moreover, it is logically consistent, because Ethernet frame carries e.g. IP packet as data, IP packet carries UDP or TCP as data, and TCP/UDP carries data of some application protocol.

Let us focus on applications running on the virtual node. From the previous discussion, one can conclude that they should receive and send data through a specific module – e.g. applications using sockets through the “Socket” module. On the other hand, those applications might want to communicate with other modules in the stack, for example application running some routing algorithm has to modify the routing tables managed probably by IP module. Hence, the applications will run independently from the linear protocol stack and they will be able to cooperate with any module, if the module provides an appropriate interface.

Figure 2 shows two examples of nodes.

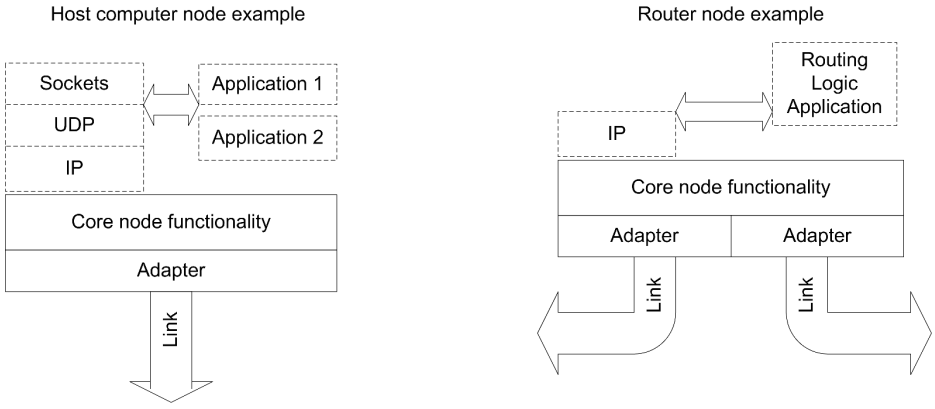


Figure 2: Examples of simulated nodes

The following chapters will explain the NetSim structure in more detail.

Simulator Internals

This chapter goes through the NetSim core components that provide basic functionality for other parts of the simulator.

Thread Management

As mentioned earlier, a good simulation behaves like a real environment. However, some behavior cannot be done exactly as it is in the real world. In a real network, many events can occur at the same time. Since the simulator runs on a computer with a (small) finite number of processors, those events cannot be simulated exactly at the same time. However, the simulation should behave almost like the real network. Doing so in a single thread would be very hard, most probably impossible, so the simulator takes advantage of threads and lets the system take care about the "simultaneous" execution.

However, using threads brings some new issues. The most complicated is probably the need of synchronization. One has to always keep in mind that there can be two or more threads accessing the same data at the same time. There is nothing about it to describe in general; almost in each component of the simulator, some kind of locking or other synchronization has to be accomplished.

Secondly, threads should be reused; otherwise, the execution will be very inefficient. Individual jobs executed by the simulator are mostly simple and short (transferring a packet from one node to another, inserting additional protocol headers to the packet, serializing packet for transfer over the real network, etc.) Creating a thread every time such task is executed would waste CPU time and operating system resources. Moreover, the job should not be executed immediately in many cases, it should be delayed some small amount of time. If a new thread was created each time, it would result in many threads just waiting.

Therefore, a tread pool of threads executing jobs from the queue is implemented. The implementation is based on Stephen Toub's ManagedThreadPool [15], which has the same interface as .NET Framework ThreadPool[8], but the queue and the way the work items are removed is modified. The queue of jobs to execute is not just the simple System.Collections.Queue as in the original solution; binary heap is used [2].

The reason why the heap (acting as a priority queue) is used has been mentioned above: timing. Work requests should not be executed in the order in which they are added, for example, a link can simulate some delay in delivery by adding a request to the thread pool to execute code that will finish the data transfer after some amount of time. It is straightforward that work requests are sorted by the time they should be executed.

[8] The same interface, but a different behavior. .NET Framework Thread Pool creates and destroys treads according to the number of work requests [3], compared to the ManagedThreadPool where constant number of threads is used. This behavior is better for the simulation since there might be high demand for data transfer after a long time of inactivity and there would be no time to create new threads.

If there is a request to do something that should be processed right now or that should have been done in the past, an available thread takes it and executes it. If there are no jobs, all threads go to sleep waiting for a new item in the queue. Finally, if there are only jobs that should be processed in the future, threads should go to sleep and wake up after the job is ready to execute. However, for performance reasons, it is better to wake up always just one thread; it than removes an item from the queue and wakes up another thread that will check the execution time of the next item and either process the job or go back to sleep for some time. The last thing to consider: a special case concerns adding a new item that should be executed before all others in the queue. In such case, the sleeping thread is woken up and it either executes the job or goes to sleep for a shorter amount of time than before.

Timer

The discussion above indicates that an accurate timing is very important for the simulation. Although the hardware usually supports some high-resolution timer, regular system timer resolution is one millisecond, which is quite a long time for today's high-speed networks. Therefore, a new timer implementation that uses this high-resolution timer is provided; only in case the hardware does not support it, a regular OS timer is used.

Since a millisecond is not an accurate time unit for the simulator, ticks (100 ns intervals) are used. The reason why ticks (and not for example microseconds) have been chosen is that .NET Framework time functions support this time unit naturally; almost all classes have Ticks property that returns the time interval in ticks.

Remote Control

Remote Control is a way for controlling a simulation programmatically by another application. This might be useful e.g. for interactive applications that visualize the network behavior and allows a user to submit commands for the simulator, or for applications that automate the simulation process. The graphic application described on page 33 also takes advantage of this feature.

There is another possibility to control the simulation – through a command line. Although it is a convenient way to instruct the simulator to do some simple actions, e.g. to stop the simulation, or enable/disable some part of the virtual network, for other tasks, mainly for retrieving information, it is a bit impractical to parse the console output. Therefore, a programmatic interface is a useful alternative to the command line.

Since .NET Remoting is used for all other inter-process communication, not surprisingly it is used also for the Remote Control. The simulator creates an object that is remotely accessible and which provides methods to affect the simulation. Moreover, a callback object can be registered and it will be called if some event in the simulator occurs; it simplifies controlling the application implementation since it does not have to examine the simulator state repeatedly.

Event Log

While the simulator is running, some events that the operator should know about can occur. For example, an application fails on some node or an attempt to redirect an application network communication via Remote Sockets to a non-existent node or a node that does not

support sockets is made. In such cases the simulator run can continue; however, it would be practical to store such information somewhere: to an event log, which can be listed on the console; or, if some application is using the Remote Control and has registered a callback object, it will be notified immediately.

Simulator Configuration

The simulator should be extensible and users should be able to add their own modules or create their own link types, which would be probably also configurable. Therefore, NetSim configuration should be designed carefully to make such extensions possible and do not force the users to make much effort to configure their own virtual networks.

Since the users should be able to add their own configurable parts, having only one configuration file with a well-defined structure is impractical; it would be better to let the users choose their preferred format[9]. For this reason, virtual network configuration consists of one main configuration file describing the initial interconnection of simulated links, nodes and their adapters, modules, and applications – just the interconnection and no details about each node, link, or module. Those details are written in separate configuration files, which are referenced from this main file (format of those files is not predefined, each module or link type can have a different one, since it is responsible for parsing its own configuration).

What is the essential information needed in the main configuration file? Obviously, a list of links and nodes, plus adapters, modules, and applications that attach to the nodes. In addition, from the previous discussion, references to other configuration files of the components. Moreover, because those files can have various formats, also some information about how to parse the data, or which component should parse it.

This is, de facto, all the information necessarily needed; all other data could be moved to separate configuration files. However, how would the case that a particular adapter is connected to a particular link be represented? There should be some unique link identification; the same applies also to the nodes. Since this identification should be the same for all components, it would be practical to have it in the main configuration file: textual string was chosen for this purpose.

It is not clear whether to have information about connections in the main configuration file or in the configuration files of the adapters. However, there are two reasons to choose the first possibility: firstly, if the adapter would be very simple (non-configurable), having a separate file to store just information about the link is a bit impractical. Secondly, having the connections in the separate files of not well-defined format will cause great problems during the implementation of a helper tool that allows for creation of the virtual network in the graphic environment.

[9] However, XML is preferred, since it is today's widely accepted format.

An example of a configuration for one node in the main configuration file follows; the complete virtual network configuration can be found in the chapter User's Manual on page 63.

```
<Node name="A" class="NetSim.Core.Node">
   <Adapter name="eth0" class="NetSim.Ethernet.EthernetAdapter"
      link="Link 4" config="nodes/A/adapters/eth0.xml" />
   <Module class="NetSim.Library.Modules.IPModule" name="IP"
      config="nodes/A/modules/IP.xml"/>
   <Module class="NetSim.Library.Modules.IcmpModule" name="ICMP" />
   <Module class="NetSim.Library.Modules.UdpModule" />
   <Module class="NetSim.Library.Modules.TcpModule" />
   <Module class="NetSim.Library.Modules.SocketModule" name="SocketModule" />
   <Application class="NetSim.Library.Applications.SimpleWebServer"
      config="nodes/A/applications/SimpleWebServer.xml"/>
</Node>
```

The *name* attribute represents a unique identification of the component. The *config* attribute is a relative path to the particular configuration file; *link* is the name of the link the adapter connects to. Finally, *class* is the name of a class that should be instantiated for a given network component (the object of this class will parse its configuration file; more information can be found in the chapter Implementation, page 24).

Other features

Interconnection to the real network

The support for data transfer between virtual and real networks is a basic presumption for the simulator usability. However, there is no need to conform the simulator design to this requirement; it can be achieved by the presented model of nodes, links, adapters, and modules. For the transfer in direction from the virtual network link to the real network, there can be an additional node with an adapter in a "promiscuous" mode attached. That adapter will pass all packets to the node core and the node to the first module in the module chain. There can be a module that will hand the packets over to a real network. Moreover, such module can listen on the real network and pass the captured packets back to the node, which will send it to the adapter and then to the link.

The NetSim implementation is done exactly as described above – a module called `ExternalConnection` is implemented. It uses the libpcap (winpcap) library and its C# wrapper SharpPcap [6].

Running Third Party Network Applications

The simulator allows a simulation of some applications by adding specific components (called *applications*) to the nodes. However, such possibility is not always sufficient – running a real application on the virtual node might be useful for example for monitoring purposes, for creating a substantive network load etc.

It would be most convenient to the users to just execute their application and configure it to be virtually running on the specified node. This is also possible with the Remote Sockets feature (see page 35).

Another way how to manage this would be to implement a socket library for virtual network and recompile the application with the new library. However, this way is a bit impractical – the compilation often requires a non-trivial build environment; moreover, the source code is unavailable in many cases. Finally, the library would be OS specific and it should be implemented for every supported operating system. (Of course, Remote Sockets is also OS specific.) Such library is not included in this work.

Finally, in case the application source code is available, in case it uses .NET sockets, and can be slightly modified, it can be easily changed to the *application* component that can be placed on the virtual node[10].

SNMP Support

It is a bit complicated to add SNMP support, because it requires information from all parts of the node, adapters, modules etc. The solution to this issue could be following: there could be a SNMP module on the node. Every part of the node (adapter, module, or application) that supports SNMP would register itself by the SNMP module and provide a set of supported MIB entries. The module would capture requests for exploring or changing values and then query the appropriate components to provide or change the values. This solution enables to implement SNMP in one module only; there would be a simple common interface for other components to provide information about them[11].

[10] The source code change yields a creation of the new class implementing `IApplication` interface, calling `Main()` from the special method of that class and replacing all occurrences of `System.Net.Socket` to `NetSimSocket`. More information about applications can be found in the chapter Implementation on page 24.

[11] SNMP support is not included in the current version. However, it is one of the priorities for the next release. See Further Work, page 52.

3. Implementation

The third chapter stems from the previous text, where the main ideas were introduced. The NetSim design is explained; some important parts are explained in more detail.

The main goal of the design is to propose a simple extensibility, which allows the simulator to be widely used. This is achieved by simulator modularity and as simple as possible interface of the modules that the users would write by themselves. To avoid any ambiguities, some vocabulary that is used in the following text should be explained now. The term *frame* always represents a link protocol packet, like Ethernet frame. Every packet (i.e. some data with headers) for all higher-level protocols (like IP) is called a *packet*.

Packets and frames

Packets and frames are maybe the simplest thing to explain – therefore it is the right part to begin. Every type of packet or frame has its own class that represents its format. Those classes inherit from an abstract class `Packet`, which allows the simulator to work without knowing all the packet types a priori.

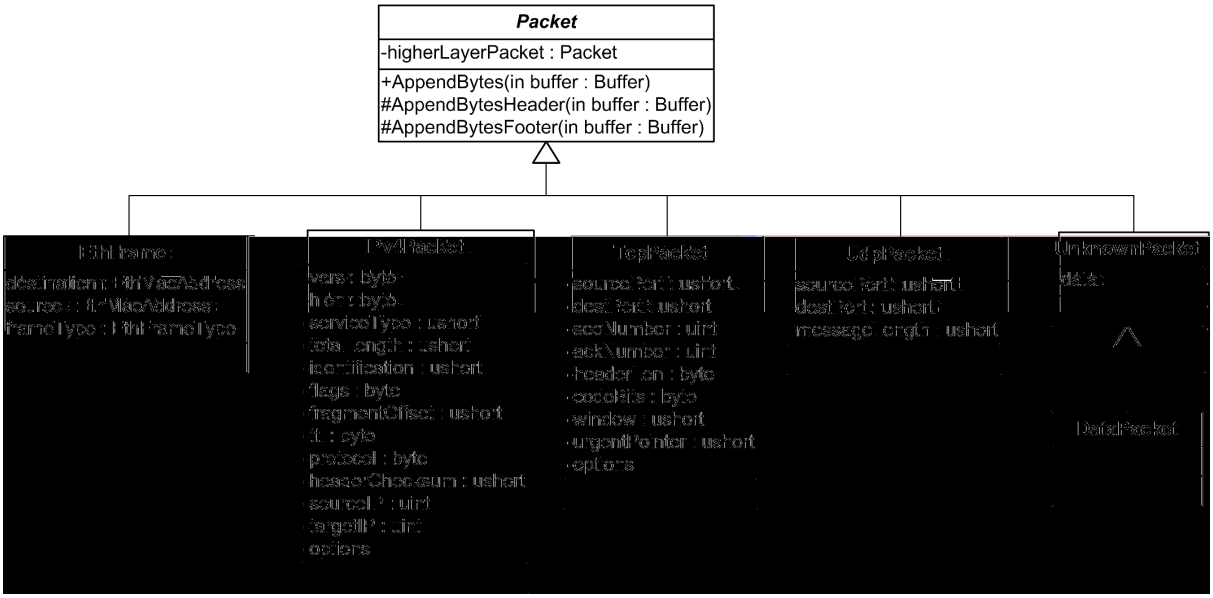


Figure 3: Packets and Frames

As depicted in Figure 3, even if there is some hierarchy between packets in a usual protocol stack, there is (almost) no inheritance hierarchy between those packets, because the combination of protocols in the protocol stack is up to the user and simulator configuration. Instead, the inheritance, packets and frames are combined into a chain, every object has a `higherLevelPacket` member that points to the object of a higher-level protocol packet (current packet data). For the packet at the end of the chain, this member is `null`.

Because the packets created and transferred through the simulated network can eventually reach a real network, there is a need to create the real network packet from the one that is used

inside the simulator. To do this effectively without copying data at each layer, every class that inherits from `Packet` has to implement `AppendBytesHeader` and `AppendBytesFooter` methods. The use of this method while creating the real network packet is illustrated in Figure 4; the data from all protocol layers are added to one buffer.

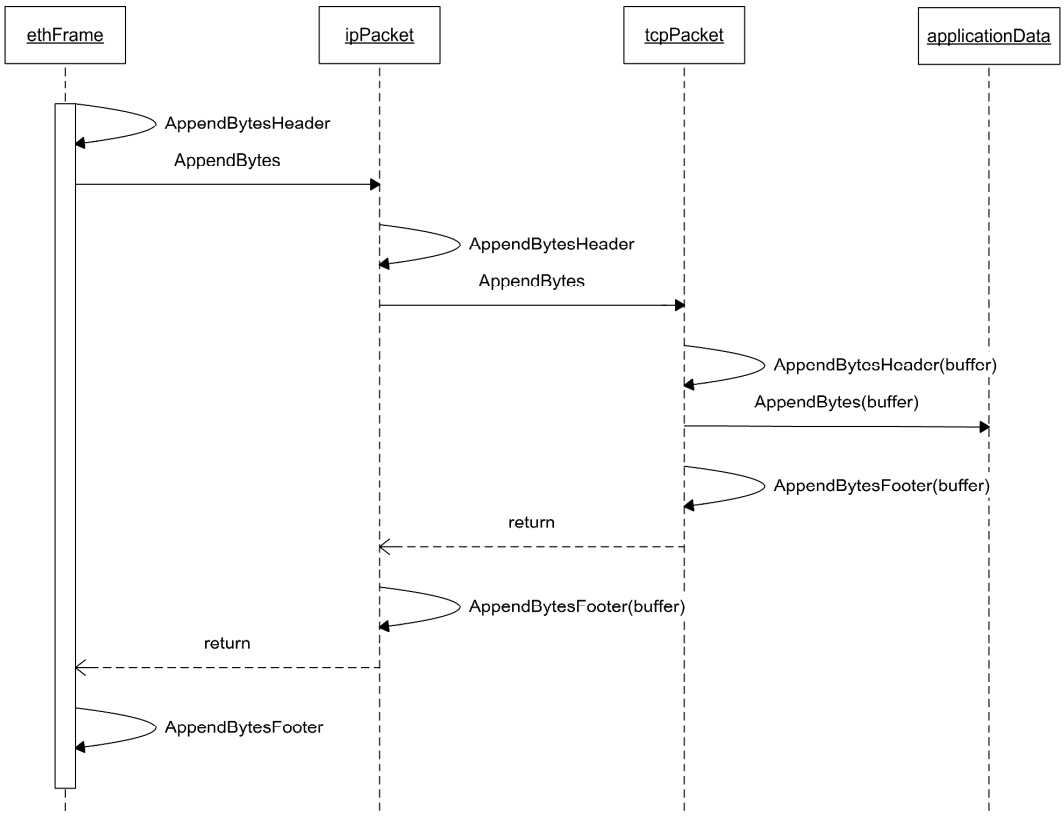


Figure 4: Creation of a binary packet for transmission over the real network

Links and Adapters

There are two abstract classes: `Link` and `Adapter`. Each link implementation has to be a descendant of `Link` (`Adapter`, respectively; for adapter, also implementing `IAdapter` interface is possible).

For transmission of packets from a node to an adapter, the adapter has to implement the method `SendPacket()`, which is called by the node. This method enqueues the packet for a future transmission. If the queue is empty and no packet is being sent, a new frame object is created according to the media used (Ethernet in our example) and passed to the link `SendPacket()` method. A successful transmission is recognized by the simplest way – the same data are received back by the adapter; whether the data are the same can be recognized just by reference comparison. If the frame cannot be sent (for example a collision occurred in Ethernet case, or the link is down), the link calls back to the adapter `PacketSendingFailed()` method. Therefore, the adapter can rely on the link to be notified about packet delivery, it does not need to implement its own timer to recognize failures. Only one queue of packets ready to send is needed – this queue is located in the adapter object. The adapter will send another frame only after a successful transmission of the previous one, so

there is no need to have a queue in the link object and no need to queue packets in the upper layers.

Receiving a frame by an adapter is similar. The link will call a `ReceivePacket()` method of the adapter; the frame has to be delivered also back to the sender as an acknowledgement of a successful transmission.

The link behavior should correspond to a simulated technology and both links and adapters can provide other media specific methods; for example, to simulate a ring based data delivery (such as Token Ring), the link should internally order the connected adapters into a ring and send the frame received from an adapter only to the next one. For bus technologies like Ethernet, the link will deliver data to all adapters at about the same time. However, in all cases, it should be specified which adapters can cooperate with the link and the link should test the adapters as they are connected and refuse the connection of an incompatible adapter.

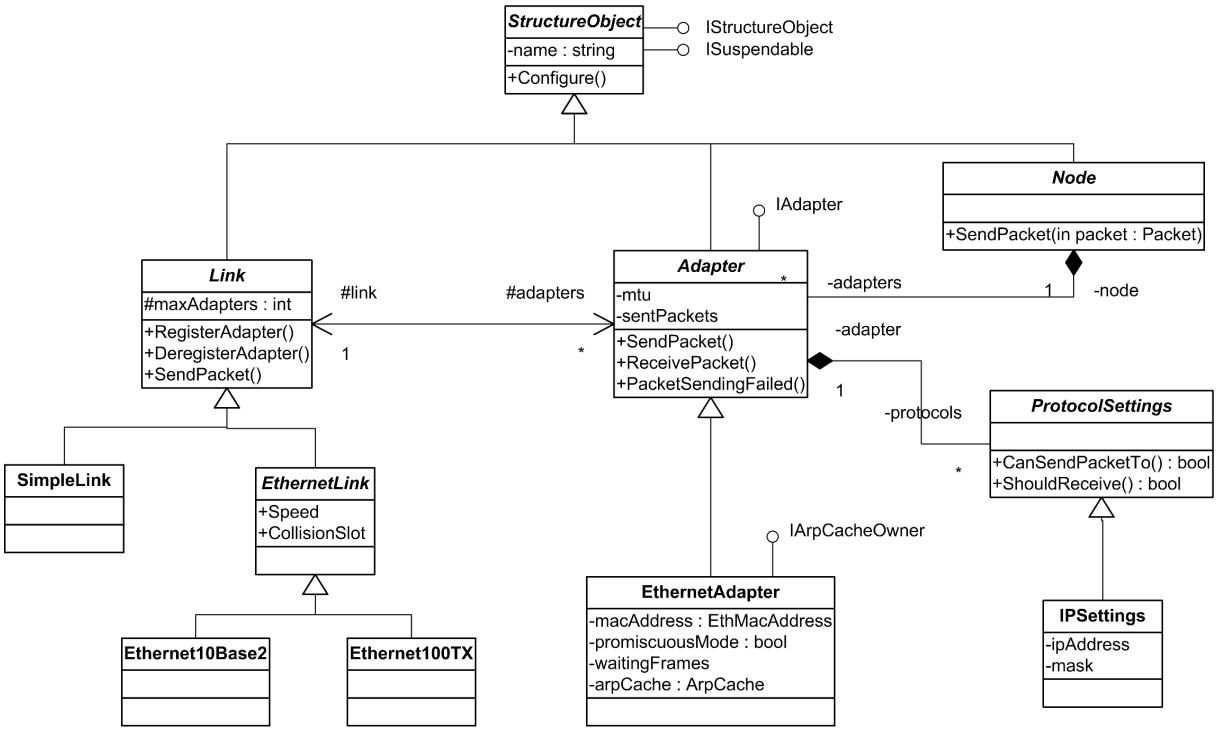


Figure 5: Links and adapters - abstract classes + Ethernet example

Figure 5 shows a class diagram of links and adapters. All adapters and links are descendants of `StructureObject` or implement `IStructureObject` interface. This ensures that all will implement a common functionality such as providing information whether enabling and disabling is available. It is shown that actual Ethernet link implementation classes derive from the `Link` class and adapter `EthernetAdapter` derives from `Adapter`.

In addition, a class `ProtocolSettings` is associated with an adapter. Only IP networks are supported in this version; however, future extension to other protocols is allowed. Therefore, it is possible to assign more than one protocol and its basic settings to the adapter. While a packet is being sent, the node should decide to which adapter the packet should be transferred. The adapter decides, according to this protocol specific information, whether it is the right one for packet delivery.

Nodes and Modules

As described in Chapter 2, nodes are associated not only with adapters, but also with *modules*, that actually provide node functionality. The node maintains a sorted list of modules that are present on the node. Whenever a frame is received through any adapter, the adapter creates a structure called `PacketInfo` that is used for passing packets up and down through the linearized protocol stack. The received frame is actually a chain of Packet class descendant objects; the adapter stores a reference to the first one to the `RawPacket` field in `PacketInfo` and a reference to the second one (that should be processed by a module) to the field `Packet`. Then, the structure is forwarded to the node. The node itself does not know how to handle packets; it just passes the structure to the first module in the list. The module should look at the `Packet` field in the structure (or also other fields); determine whether it is a known packet type and whether it should be processed. If not, it does not do anything and the node will pass it to the next module.

In case the module should process the packet, it can either set the reference to the whole `PacketInfo` to `null`, which means that there is no need to continue in forwarding the packet up the protocol stack; for example, if the IP module recognizes that the packet target is not a current node and performs routing to the next hop destination. Second, the fields in `PacketInfo` can be just changed; mostly it is the member `Packet`, which is changed to point to the packet object of the next layer.

An example of packet receiving is depicted in Figure 6; the objects in dotted boxes are always the same, of course, they are shown multiple times to make the figure more understandable. Thanks to the `RawPacket` reference, any module can have complete information about the frame received; therefore, an Ethernet switch can be implemented as a module and there is no need to have different types for nodes that operate at link layer.

Any module can also send a packet; it creates its own `PacketInfo` and calls `SendPacket()` method on the node. The node then passes the packet info down to all modules layered below the sending one and they can change the packet being sent appropriately – mostly new packet objects that contain low level packet headers will be added to the chain. If some module sets `RawPacket` while the packet is being sent, it forces the adapter not to add its own headers and take the packet as a raw frame; this allows modules to have direct access to the link media. However, in such case, the module should exactly know the links the node is connected to – this would be used probably for nodes operating at link layer, such as previously mentioned Ethernet switch.

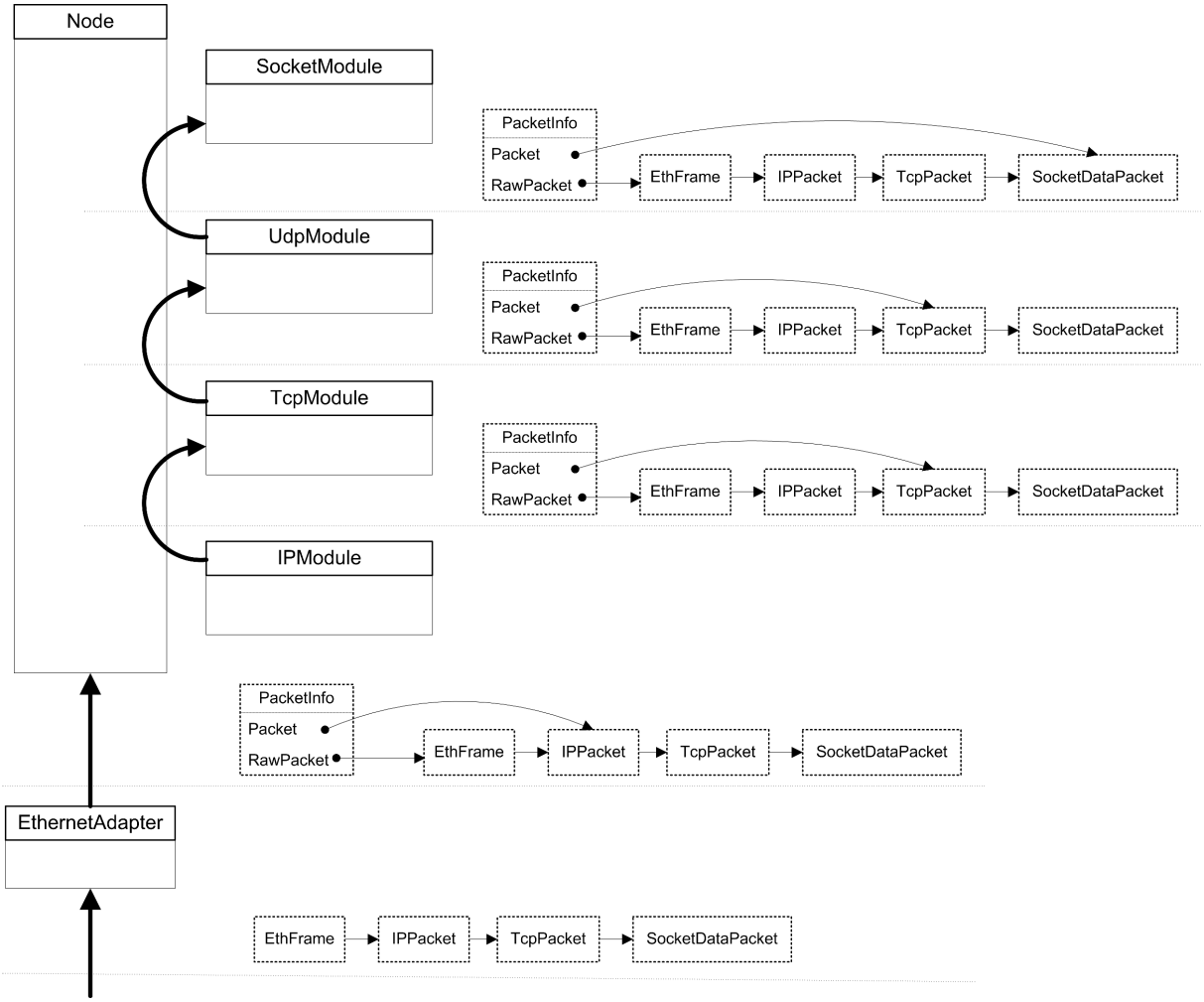


Figure 6: Passing received packet up the protocol stack

The modules have to either inherit from an abstract class `Module` or implement `IModule` interface to allow the node handle any type of module. Figure 7 shows some examples of modules that are implemented and the relationships between `Module`, `Node`, and `StructureObject` classes.

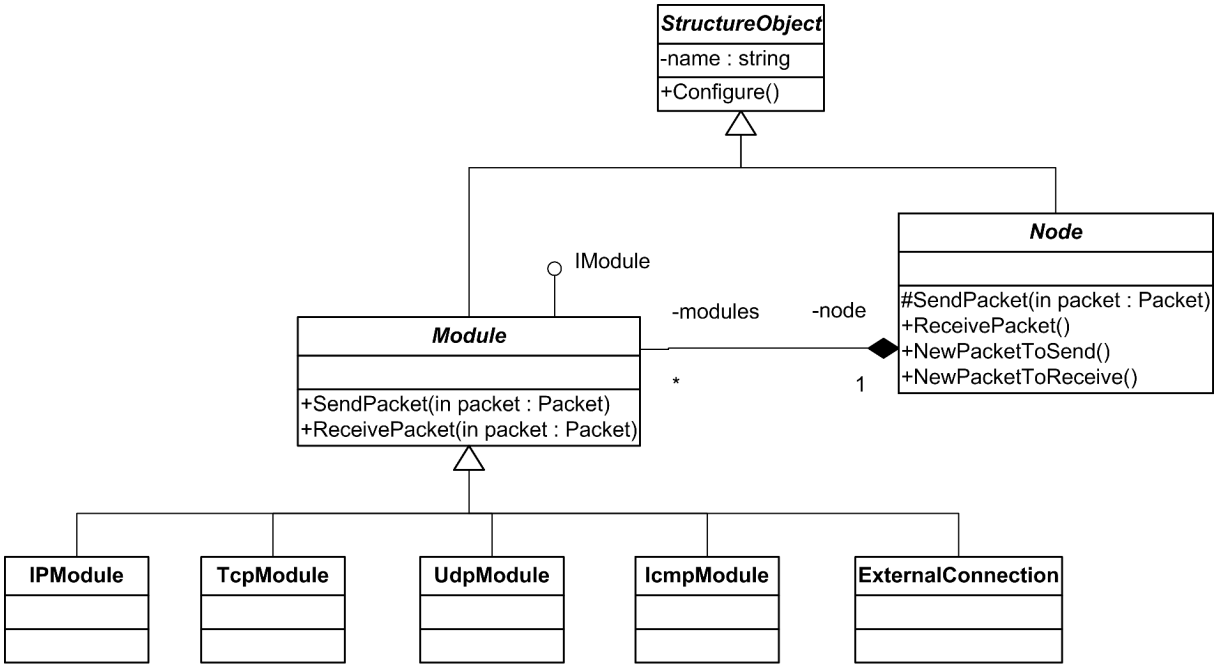


Figure 7: Examples of modules and their relationship to the node

Modules can interact each other; for example, when `IPModule` receives a packet that should be forwarded to an unknown destination, an ICMP message should be generated and sent back to the sender. However, a separate module called `IcmpModule` handles ICMP messages; therefore, `IPModule` instructs it to send a 'Destination Unreachable' message. Nevertheless, how can the `IPModule` reach ICMP, if only the node has a list of references? Each module can have a name, which is specified in the virtual network configuration file, and other modules can ask the node to obtain a reference to the module of such name. Although the reference can be obtained, modules should not store the reference for future use and should ask for it each time they need it, since the node configuration could change and the previously obtained reference may be invalid[12].

Modules also lie in between the communication of applications and the node. For example, when an application creates a socket, it registers itself in the `SocketModule` and all communication goes through this module.

Applications

Applications are an important part of a network structure since they generate traffic sent over the network. NetSim provides two possibilities to allow an application to run on the specific virtual node. The first one called Remote Sockets is intended for the existing real applications without recompilation; Chapter 4 describes it in detail. The second possibility is writing a new application or modifying an existing one to run on the virtual network.

[12] In future releases, the node could return reference to a proxy object of a module, which will know that its module has been deleted.

Writing an application specifically for NetSim involves creation of a subclass of `NodeApplication` or implementing `INodeApplication` interface. When inheriting `NodeApplication`, `Run()` method, which is called from `Start()`, should be overridden. `Start()` just encapsulates `Run()` by error checking and letting the simulator know about uncaught exceptions that occur in the application. When implementing an interface, just `Start()` is needed. Figure 8 shows the `NodeApplication` class, some descendants implemented in the NetSim library and their relationship to the node.

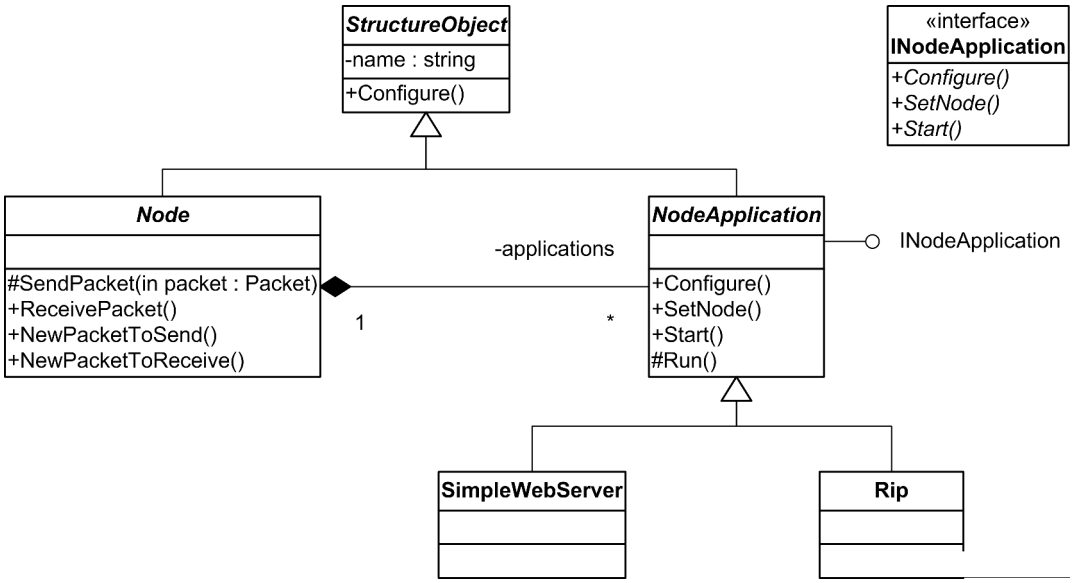


Figure 8: Examples of applications and their relationship to the node

While the simulator is starting, it creates a separate thread for each application and executes the `Start()` method, which is like `Main()` for a regular application. Then, if the application needs more threads, it can create them as usual; however, they should be registered by the `NodeApplication` object. This is necessary for the simulator to be able to pause/resume the simulation process. If the pause is requested, all threads, including the applications, should be suspended, because the application could try to send data and report errors to the user (that data would remain queued and would not be delivered).

Libraries

The previous chapters described the core of the simulator. However, for practical use also a set of functionality is required. As it is not possible for every user to create his/her own nodes, links, and protocols implementation, NetSim provides a standard library. Ethernet library is separated as a small example showing how to extend the simulator with a new functionality. (Some classes included in those libraries were shown in previous figures.)

While the simulator is starting, it loads the libraries according to a configuration file. The main configuration file can contain for example the following elements:

```
<Extension path="<path>/NetSimEthernet.dll">
   <PacketFactory class="NetSim.Ethernet.EthernetPacketFactory" />
</Extension>
<Extension path="<path>/NetSimLibrary.dll">
   <PacketFactory class="NetSim.Library.LibPacketFactory" />
</Extension>
```

This tells NetSim to load two libraries. Those libraries are then searched for classes specified further in the configuration file. The search is performed sequentially in the order the extensions are specified; it is possible to use some functionality from the second library and some replace by the first library. However, this is not a recommended procedure, using different class names or different namespace in each library is much better technique.

Every extension can have any number of *Packet Factories*. If the extension adds a new type of protocol, it probably needs to use new packet type. As soon as the simulator runs separately from the real network, those new types of packets are created/sent/received only by this extension, which knows their structure. However, if those types of packets come from the live network, they are just a sequence of bytes and the simulator has to know how to create `Packet` object from that sequence. The simulator (usually some module) knows the network layer (in the sense of ISO/OSI model) at which the packet is used and some identification from the lower layer[13]. This is the only information available for packet identification.

Therefore, when loaded, the packet factory tells NetSim the types of packets it can create: the ISO/OSI layer and packet type identification. When needed, the simulator then asks the factory to create packet object from an array of bytes.

The description of classes implemented in the libraries distributed as a part of the NetSim package follows. Great Comer's book about TCP/IP and adequate RFC documents have been used as a reference for the implementation [4].

Standard NetSim Library

This library provides implementation of core components that are needed for practical use of the simulator; it will be definitely enlarged in future versions. The currently implemented features include: TCP/IP protocols, RIP routing, interconnection to the real network, and `SocketModule` used for data transfer between virtual node and `NetSimSocket` (a socket that can be used by applications running on virtual nodes and provides the same interface as standard system socket).

TCP/IP protocol suite

The support of TCP/IP includes the implementation of packets, packet factories, and especially modules that support for example routing, resending, or sorting of out of order delivered packets.

[13] That identification coincides with identification numbers assigned by IANA organization. The numbers assigned can be found at http://www.iana.org/assignments/protocol-numbers

`IPModule` is the basic module in TCP/IP implementation. It handles the core IP functionality: fragmentation, defragmentation, and routing. It maintains a routing table internally; entries can be added either via the configuration file, or dynamically by other parts of the node while the simulator is running (see RIP below). It also uses `IcmpModule` to send ICMP messages if some error should be reported to remote node.

`IcmpModule` should be layered over IP; it is capable of sending and receiving ICMP messages. Either reacts to incoming messages (ping), or provides support for other modules or applications for error reporting to remote destinations.

`UdpModule` is quite a simple module implementing UDP protocol.

Compared to UDP, `TcpModule` is a complex implementation of TCP protocol. It provides support for establishing connection and disconnecting, resending data, sorting of data that came out of order.

RIP

Currently only the second version of the RIP protocol is supported. It is covered by a single application that takes advantage of a socket support in NetSim. The application listens on the port specified in the configuration file and periodically sends a routing table to neighbor routers (also from the configuration file).

It cooperates with an `IPModule`, where the routing table is stored, reads and changes the routing information appropriately.

External Connection

This is a single module `ExternalConnection`, which internally uses WinPcap (or libpcap on Linux) to send and receive packets. Currently the wrapper SharpPcap is used [6], but it was modified and many code remains unused; an exclusion of SharpPcap and using packet capture library directly is one of the tasks in the TODO list.

While the simulator is configured, the capture library is initialized; the device where it should listen is specified and opened. Then, at the time the simulator is started, the packet capturing is started too. When a packet arrives, it is parsed, the simulator internal representation of the packet is created and it is sent to the virtual network. On the other hand, if the packet is received by the module from the virtual network, an array of bytes is created and sent through the packet capture library to the real network.

This module is intended to be the only one module on the node. For a description of how to interconnect virtual and live networks see the User's Manual in Appendix C, page 60.

Ethernet Library

Ethernet library could be a part of the standard library; however, it is intended to remain as a small example library, even if the standard library will grow. It is a good example for the developers that will add their own libraries, since it shows addition of a new protocol with packet factory and one simple module (for Ethernet switching).

To support Ethernet functionality fully, an adapter (`EthernetAdapter`), link (`EthernetLink`), frame (`EthFrame`), and address (`EthMacAddress`) are implemented; all of them are the

descendants of particular abstract classes intended for extending the set of the network components.

`EthFrame` is a new packet (frame, actually) type; along with the type, its factory that should be registered with the simulator is implemented.

`EthernetLink` provides a common algorithm for delivery, such as applying delay and collision recognition if two frames are sent at one time. Two other classes are further inherited from that class, they differ just by the link speed (delay after the frames are sent further): Ethernet10Base2 and Ethernet100TX.

`EthernetAdapter` wraps incoming packets to the `EthFrame` and sends – it does not need to be connected exactly to the `EthernetLink`; however, resending with exponential back-off is included, so it works well together with collision detection implemented by the link; exactly as the real Ethernet. The adapter has also its own `ArpCache` object that stores the upper-layer address to MAC address mapping. The adapter itself sends and receives ARP packets and sends the request automatically if the packet to an unknown destination address should be sent.

`EthMacAddress` is an address implementation allowing the simulator to handle the Ethernet addresses easily via its parent class `NetworkAddress`.

Finally, a module `LearningSwitch` implements the behavior of a cheap non-configurable Ethernet learning switch.

User Interface

The previous chapters described the internal simulator components; however, how nicely the simulator engine is designed and implemented is maybe not so important for the users, as they will appreciate a subtle and usable interface. There are both console and graphic UI available; however, the graphic is implemented currently for Windows only.

Console

Console application is very simple; it just parses command line arguments, loads simulator assembly, configures the simulator and prompts the user to write commands. Currently only enabling/disabling of the simulator components is available (plus printing some information), the network structure cannot be changed through the console commands. Changing the configuration scripts directly is needed while using the console only. However, a set of scripts generated by the graphic application also can be used (since GUI uses console internally).

Graphic Application

Graphic application (GUI) is both a network designer and a tool for running simulations. It allows a creation of the simulated network by simply using drag&drop from the panel of predefined network components. The application is also designed to be extensible, new components can be added by modification of a configuration file that is checked each time the application starts and all predefined components are added to the panels of nodes and links (a screenshot is shown in Figure 23 on page 63).

GUI maintains a main configuration file, since its format is known, and therefore it can be modified according to the changes to the virtual network made by the user in the designer. On the other hand, configuration files for separate network components cannot be simply modified since their format is generally unknown; however, the GUI is capable of starting a user-defined editor for modifying that configuration files.

To provide a better application stability, the simulator runs within a separate process; therefore, if there would be some error in the simulator that would make the process unstable, the graphic application will continue and inform the user about such bad event. For this purpose, GUI runs the console application internally and some commands are sent directly to its standard input. (Moreover, the console is also available to the user within the GUI.)

Nevertheless, not all communication between GUI and the simulator is done via the console, a connection via .NET Remoting is also used (see Remote Control, page 20).

More information on using the simulator can be found in Appendix C, page 60.

4. Remote Sockets

This chapter introduces the Remote Sockets feature, which is unique for NetSim and is not present in other existing network simulators. Remote Sockets allows redirection of any application network communication to any node in the virtual network. It is obvious that this feature is operating system specific; this work focuses only on MS Windows. Support for other operating systems might be added in the future[14].

Redirection of an application communication requires interception of a way in which data are delivered between the application and a network card driver. Windows provide two points where it is possible to insert a component that all data will go through it.

First, it is the NDIS[15] kernel driver. NDIS allows for creation of three types of drivers: Miniport, Intermediate, and Protocol. Miniport driver provides communication between hardware (network card) and NDIS library. Protocol drivers transfer data between NDIS and upper layer protocol (via private interface). Intermediate drivers are layered in between and can have any purpose, e.g. data filtering or, in our case, data redirection [10]. Described architecture is shown in Figure 9.

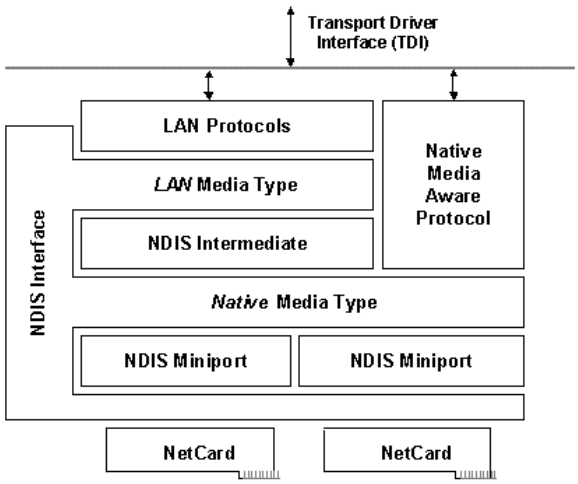


Figure 9: NDIS Driver types and layering[16]

However, this implementation brings many issues. Firstly, the need of kernel driver development and then its installation, which requires administrator privileges and many administrators prefer to avoid the installation of additional drivers (especially those not signed by Microsoft). Secondly, protocol headers changes. While the data goes through the protocol stack, headers containing addresses and other information are added to form packets.

[14] For Linux, LD_PRELOAD environment variable can cause loading different library than the standard one, which can replace socket function calls and therefore redirect network data transfer.

[15] NDIS stands for Network Driver Interface Specification.

[16] The figure was taken from MSDN Library.

However, the local machine address is used as the sender address and in many cases also the destination will differ from the one that should be assigned in the virtual network. Thirdly, performance issues; there would be a lot of work done for nothing. Headers added by the system protocol stack would have to be changed, checksums recomputed. The advantage of this approach is quite simple interface to the upper and lower layer; just “packet is coming” and “send this bunch of data”.

Second possible point of interception is the protocol stack itself. Windows Sockets allows changes to the stack by so called WinSock Catalog, which is maintained by the system and describing installed protocol chains. When a socket is being created, the system determines which chain is the best according to the address family, socket type, and protocol specified.

One entry in the protocol chain is called service provider; two types of service providers can reside in a chain: base provider and layered service provider. Base provider is always the lowest one and provides communication with the NDIS protocol driver. Any number of layered providers can be installed over the base provider – they can provide for example authentication or security [18]. In the case of NetSim, the provider would redirect WinSock calls to NetSim implementation redirecting them to the virtual network. This solution eliminates all disadvantages mentioned in the first case, but there is one main drawback: a necessity to implement a huge WSP interface. However, this is the solution chosen for this work.

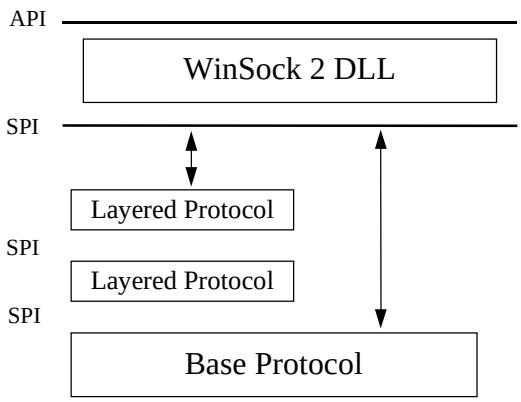


Figure 10: WinSock Protocol Chain[17]

Layered Service Provider

As mentioned above, Layered Service Provider that will be installed over other providers is implemented. Microsoft provides an example of simple layered provider in its Platform SDK[18]; a LSP for redirecting network communication is based on that example.

[17] The figure was taken from WinSock WSP Interface documentation.

[18] Windows Server 2003 SP1 Platform SDK was used for the development; there is currently new release named Windows Server 2003 R2 Platform SDK available. Both of them can be downloaded at http://www.microsoft.com/downloads/

First, some basic background how the LSP works should be presented. When the application is about to use network communication through WinSock, it calls `socket()` function to create a new socket. This forces the WinSock library to be loaded; it determines which protocol chain is the best for such type of socket and loads appropriate DLLs for providers in the chain. Provider's DLLs are regular native unmanaged libraries; `DllMain()` is called to initialize the library. Then, WinSock invokes `WSPStartup()` to determine some information about the provider and addresses of other provider functions; it also provides a table of function pointers to its own functions that can be called by the provider. During the initialization process, if the provider is a layered provider, it initializes the lower layer provider, gets its set of functions etc.

Because the layered providers' DLL is loaded into the original process address space, some kind of inter-process communication with the simulator is needed. Moreover, when installed, the LSP will be loaded and used in each application using network communication; obviously, only some set of applications should be redirected. Therefore, LSP requires also information which application to redirect.

Since NetSim is written for .NET framework in C#, its 'natural' inter-process communication is .NET Remoting. It is probably also the only choice if the simulator should remain completely managed and independent of unmanaged code (for accessing Windows API); therefore, the LSP has to use .NET Remoting too. However, it is not so straightforward. Using remoting requires using managed code, but we have mentioned that providers' DLL is unmanaged. This implies the need of mixed DLL; using just Platform Invoke[19] is not enough since calling managed code from unmanaged is required. One issue arises: mixed DLL requires a managed runtime initialization function to be called as the first of all functions in the library. However, because the LSP DLL is loaded by WinSock, this requirement cannot be met. A solution is to write two DLL libraries: one unmanaged loaded by the WinSock (it is called LSP in this document) and one mixed loaded by the LSP (called LSPWrapper).

Redirection within the LSP

The Microsoft provided LSP example creates an internal structure for each socket; this structure contains also a pointer to another structure with the information about the lower layer provider, which then contains a table of function pointers used to access lower layer providers' interface. This behavior can be simply availed: while creating the socket, if the communication should be redirected[20], the pointer to the providers' info can be changed to different structure created for the purpose of redirection. After the socket is created, no function call using that socket will call the functions of the lower layer provider, but different ones that will redirect the communication to NetSim.

---

[19] Mixed DLL means one DLL including both managed and unmanaged code. Platform Invoke is .NET feature for simplifying calls from managed code to unmanaged, parameters and return values are converted (marshaled) automatically. For more information, see MSDN library.

[20] For the information which application should be redirected, shared memory and process ID is used. See Redirection Manager, page 39.

Making the redirection in that manner has two advantages. Firstly, detecting whether to redirect or not is done only once for each socket. Additional usage of the socket, e.g. for data transfer, is redirected automatically. Secondly, the solution is robust, because it is impossible to redirect only a part of the communication, so the lower layer provider cannot become confused about a function call for a socket that had not opened.

Communication with the Simulator

As previously mentioned, communication between LSPWrapper and NetSim is done via .NET Remoting. Although it allows communication on one computer only, currently Shared Memory Channel (`ShmChannel`) is used [13]. TCP channel would be much more flexible, allowing an application to run on a different computer than the simulator. However, using a channel that requires network communication is a non-trivial task, because the redirection of one network access would cause another network communication (within the same process) and that additional network access should not be redirected. Simple global counter of provider function calls fails, because the application can use more threads to access single socket. Per-thread counter is not the right solution too, since .NET internally creates some additional threads to perform data transfer. Adding the possibility to run the application on a different machine than the simulator is one of the tasks in the TODO list (see Further Work, page 52).

The description of the method the simulator is accessed follows. When an LSPWrapper is loaded, it registers `ShmChannel` for further usage. Then the wrapper waits for the `socket()` call by the application, which communication should be redirected to the simulator. At this time, it tries to obtain remote reference to `RemoteSocketsManager` object; that object manages remotely created sockets in NetSim. If it fails, `socket()` function returns error, otherwise the remote object (residing in the simulator address space) is called, creates `NetSimSocket` and returns some internal identification of such socket, which is then returned to the calling application. For all additional accesses to that socket, the same reference to the `RemoteSocketsManager` is used, since `RemoteSocketsManager` lifetime never expires.

Callbacks

Obviously, sometimes the simulator needs to call an application, for example if an asynchronous receive operation has been requested and new data came through the virtual network. For such case, an object that can be remotely accessed should reside in the LSPWrapper. There can be only one such object per process, or one object for each created socket. Because the number of sockets created by the application is usually not large, the second option is implemented. It eliminates the need of transferring socket identification and than lookup for the information about the socket. (However, that table of opened sockets exists since only the identification of a socket comes when LSP is called by the application.)

A structure of LSP, LSPWrapper, and the simulator communication is shown in Figure 11.

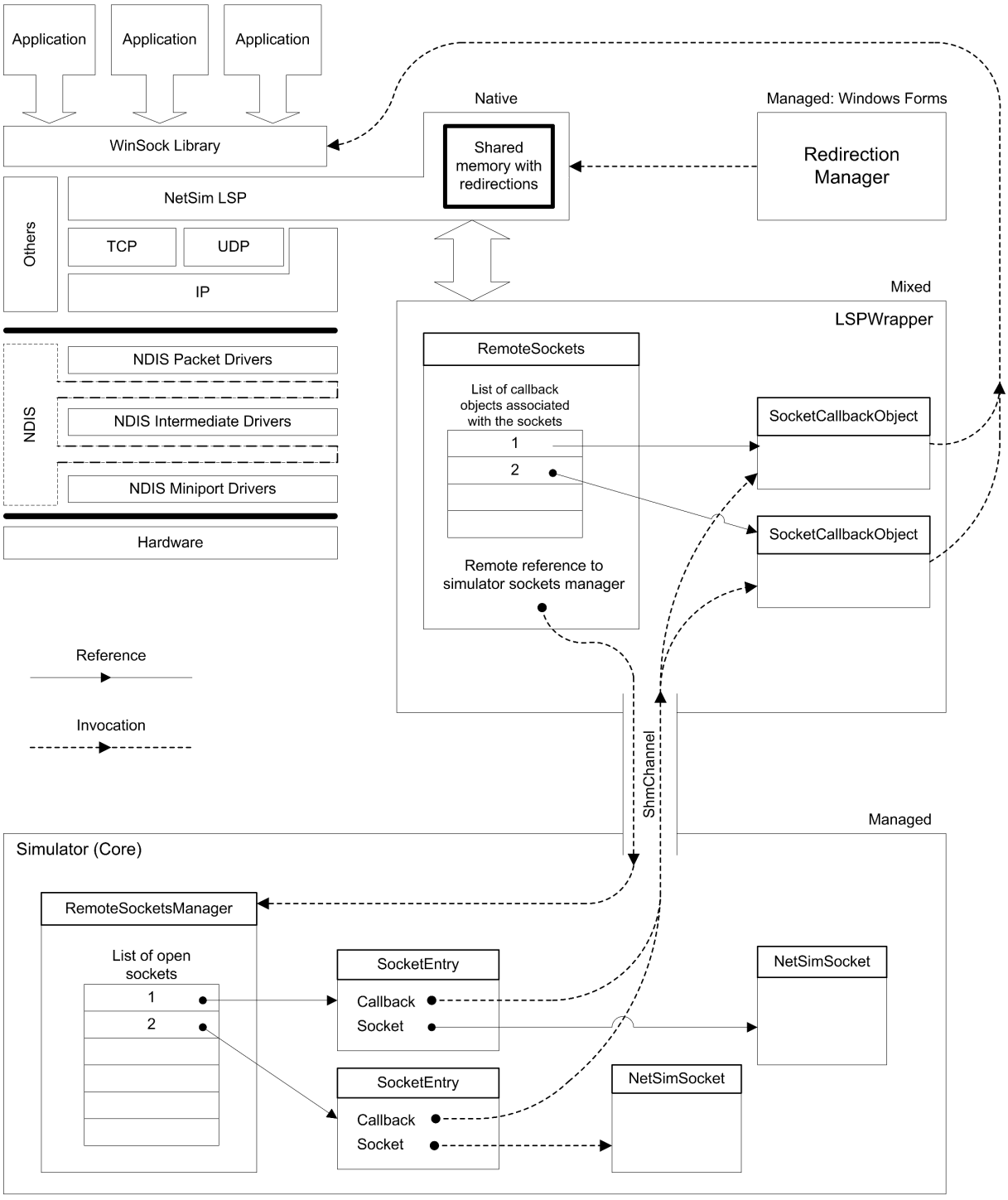


Figure 11: LSP, LSPWrapper, and simulator interaction

Redirection Manager

Finally, the way of letting the LSP know which communication should be redirected and which should pass through to the lower layer shall be discussed.

Realize that such information should be common for all applications that use or would possibly use Windows sockets. Moreover, the access to such information should be fast, otherwise it will slow down all applications that uses the network. Because of those two

requirements, shared memory in LSP was chosen. Currently its size is fixed and therefore the maximum number of redirected applications cannot exceed 32 (of course, the value can be changed, but currently in the source code only).

What is the information in that shared memory? First, an identification of an application. Because we need a unique per computer identification, process ID is probably the best choice, which is commonly used in such situations. Second, a virtual node to which the communication should be redirected. Since node names are unique in the simulator, node name is adequate. Although this is currently enough (only the redirection to one instance of simulator running on local computer is supported), server IP address and port have been added for future use, allowing more than one instance of simulator running on different computers.

To allow modifying a set of applications (processes) being redirected, a small graphic application called Redirection Manager was developed. It simply loads LSP into its address space and modifies the shared information for redirecting, so all processes that use WinSock know about the change, because they have loaded LSP DLL. That application allows displaying currently valid redirections, adding new redirections based on process ID, running new application and redirecting its communication immediately. A screenshot is shown in Figure 12; console application for this purpose is not created yet; however, its implementation is planned (see Further Work, page 52).

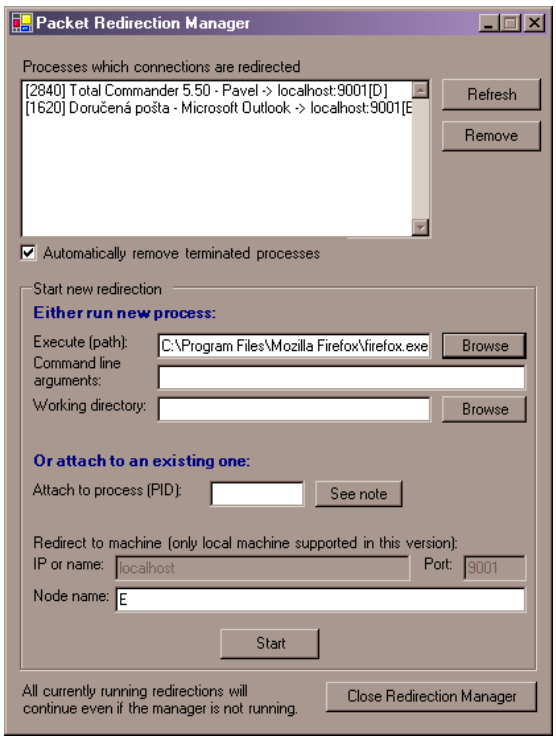


Figure 12: Redirection Manager

Note to the Implementation

In this version, the Remote Sockets feature is marked as EXPERIMENTAL, because the code is not finished yet, nor well tested. However, testing of some working scenarios has shown that ideas described in the previous paragraphs are applicable and the redirection is fast enough for practical use. Examples of applications currently working with this version of Remote Sockets are Telnet or WinSCP.

5. Practical Tests

In this chapter, some examples of the NetSim usage are shown; this should demonstrate a practical usability of the current version.

A machine running NetSim is always the same: CPU AMD Athlon XP 1800+ (1.54 GHz), 1 GB Memory (DDR2, 400MHz), Windows XP SP2. All network connections are 100 MBit Ethernet. Other computers do not do anything sophisticated; they are fast enough to process the incoming or outgoing traffic; the speed of network connection is a limitation for them.

Using Virtual Nodes from Live Network

The following text will demonstrate that the virtual network actually behaves like real to other computers. The situation is shown in Figure 13.

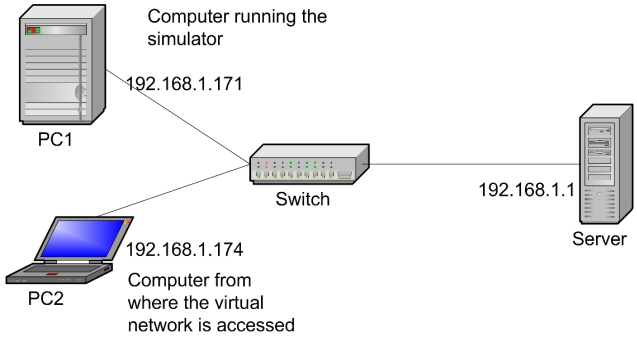


Figure 13: Accessing virtual network test

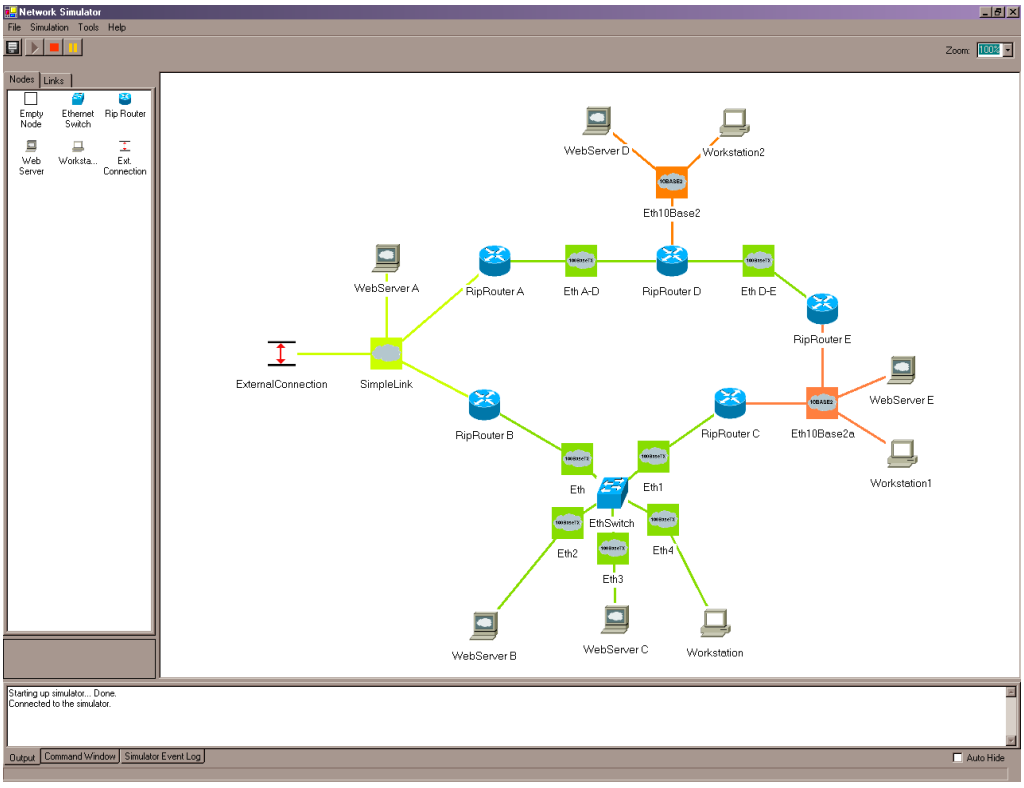


Figure 14: Virtual network in the NetSim GUI application

PC1 is running the virtual network that is provided as an example in the NetSim package. Figure 14 shows its design in the NetSim GUI, Figure 15 then its structure along with the connections to the real computers.

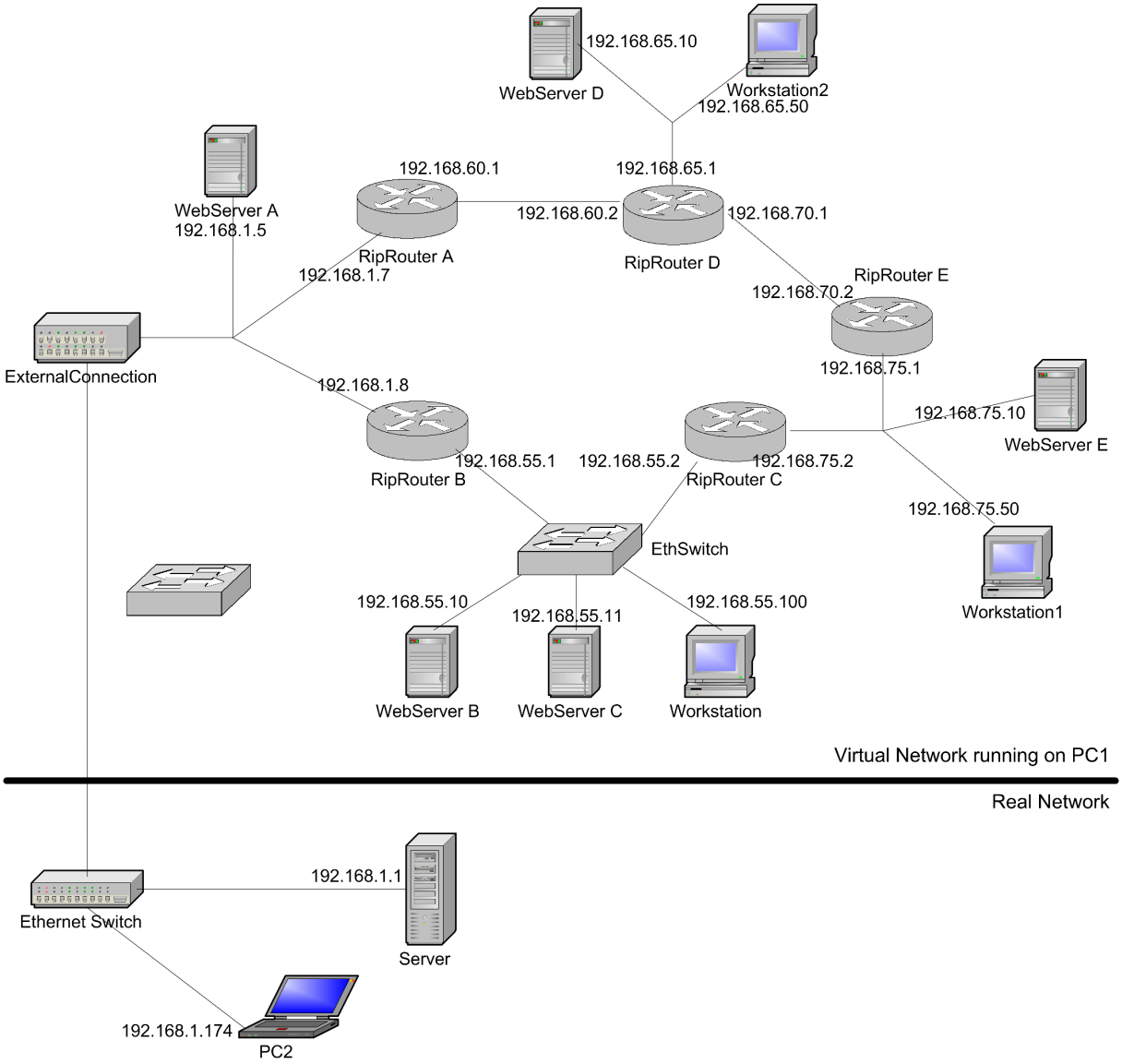


Figure 15: Network structure for the accessing virtual netwok test

The Server is a regular Linux server; RIP routing is enabled. It has set up two virtual routers that attach directly to the real network as neighbor routers that should receive routing information; those routers also have the Server's address in their RIP configuration file. PC2 will run the applications that will access the virtual network on PC1. Moreover, it has set Server's IP address as a default gateway, therefore the RIP on virtual nodes should be running properly to enable PC2 and Server access all virtual nodes.

The first test was just `ping` to the virtual network; it should be working and the return time should be quite stable, as it will be in the real network with low traffic. Ping was performed from PC2 to the "RipRouter E" node (192.168.70.2).

The ping test has been made twice, first with 512 bytes of data, second with 50,000 bytes of data. Times of the first one vary from 7 to 12 ms, times of the second one from 79 to 86 ms. It should be noticed that the same ping of 50,000 bytes directly from PC2 to the Sever is about 20 ms. The original console output follows.

Pinging the “RipRouter E” virtual node (over 2 virtual hops) with 512 bytes of data:

```
C:\>ping -t 192.168.70.2 -l 512

Pinging 192.168.70.2 with 512 bytes of data:

Reply from 192.168.70.2: bytes=512 time=10ms TTL=126
Reply from 192.168.70.2: bytes=512 time=8ms TTL=126
Reply from 192.168.70.2: bytes=512 time=12ms TTL=126
Reply from 192.168.70.2: bytes=512 time=10ms TTL=126
Reply from 192.168.70.2: bytes=512 time=9ms TTL=126
Reply from 192.168.70.2: bytes=512 time=8ms TTL=126
Reply from 192.168.70.2: bytes=512 time=12ms TTL=126
Reply from 192.168.70.2: bytes=512 time=10ms TTL=126
Reply from 192.168.70.2: bytes=512 time=9ms TTL=126
Reply from 192.168.70.2: bytes=512 time=7ms TTL=126
Reply from 192.168.70.2: bytes=512 time=11ms TTL=126
Reply from 192.168.70.2: bytes=512 time=10ms TTL=126
Reply from 192.168.70.2: bytes=512 time=8ms TTL=126
Reply from 192.168.70.2: bytes=512 time=13ms TTL=126
Reply from 192.168.70.2: bytes=512 time=11ms TTL=126
Reply from 192.168.70.2: bytes=512 time=10ms TTL=126
Reply from 192.168.70.2: bytes=512 time=8ms TTL=126
Reply from 192.168.70.2: bytes=512 time=12ms TTL=126
Reply from 192.168.70.2: bytes=512 time=11ms TTL=126
Reply from 192.168.70.2: bytes=512 time=9ms TTL=126
Reply from 192.168.70.2: bytes=512 time=8ms TTL=126
Reply from 192.168.70.2: bytes=512 time=12ms TTL=126
Reply from 192.168.70.2: bytes=512 time=10ms TTL=126
Reply from 192.168.70.2: bytes=512 time=9ms TTL=126

Ping statistics for 192.168.70.2:
    Packets: Sent = 24, Received = 24, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 7ms, Maximum = 13ms, Average = 9ms
```

Similar to above, but with 50,000 bytes of data:

```
C:\>ping -t 192.168.70.2 -l 50000

Pinging 192.168.70.2 with 50000 bytes of data:

Reply from 192.168.70.2: bytes=50000 time=86ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=82ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=83ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=79ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=83ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=80ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=81ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=81ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=81ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=82ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=80ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=81ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=80ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=81ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=80ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=81ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=80ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=82ms TTL=126
Reply from 192.168.70.2: bytes=50000 time=80ms TTL=126

Ping statistics for 192.168.70.2:
    Packets: Sent = 19, Received = 19, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 79ms, Maximum = 86ms, Average = 81ms
```

Next, the traceroute command was executed:

```
C:\>tracert 192.168.70.2

Tracing route to 192.168.70.2 over a maximum of 30 hops

  1    <1 ms    <1 ms    <1 ms  192.168.1.1
  2    13 ms    15 ms    15 ms  192.168.1.7
  3    14 ms    15 ms    15 ms  192.168.60.2
  4    14 ms    15 ms    15 ms  192.168.70.2

Trace complete.
```

After this trace, “RipRouter A” node was disabled. Since the previous route goes through that node (192.168.1.7), the traceroute commands executed afterwards was not able to reach the destination. However, after few minutes RIP updated routing tables and an additional route was found, as the next traceroute output demonstrates:

```
C:\>tracert 192.168.70.2

Tracing route to 192.168.70.2 over a maximum of 30 hops

  1    <1 ms    <1 ms    <1 ms  192.168.1.1
  2    10 ms    15 ms    14 ms  192.168.1.8
  3    10 ms    15 ms    15 ms  192.168.55.2
  4    10 ms    15 ms    15 ms  192.168.70.2

Trace complete.
```

The traceroute output shows also a bottleneck of the data transfer – that is a transfer of packets between real and virtual network. Even though it causes delay only, it is fast enough to transfer a lot of data, as will be demonstrated in the following section. Nevertheless, a proposed task in the TODO list, excluding SharpPcap and using packet capture library directly, could make these numbers a bit better.

A final test tried `SimpleWebServer` module running on “WebServer A”. As expected, the web browser displayed the following page according to the SimpleWebServer application configuration file:

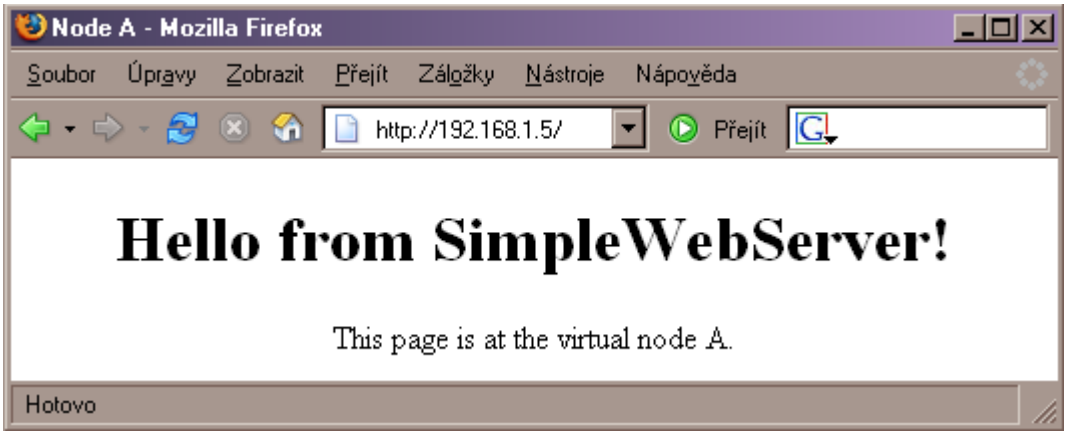


Figure 16: SimpleWebServer page in a web browser

Interconnecting Real Nodes through the Simulated Network

The goal of this part is to test how fast the data can flow through the virtual network. A schema of the real network is in Figure 17.

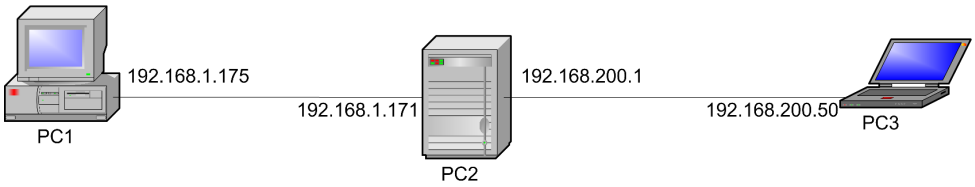


Figure 17: Testing virtual network throughput

PC2 is running NetSim; it has two physical interfaces, one connects to the PC1 and the second one to the PC3. The virtual network has two "ExternalConnection" nodes; the first one captures and sends files to the interface connected to PC1, the second one to the other interface. Data are transferred via four routers in the virtual network; its design in NetSim GUI shows Figure 18, complete network structure including both real and virtual networks is depicted in Figure 19.

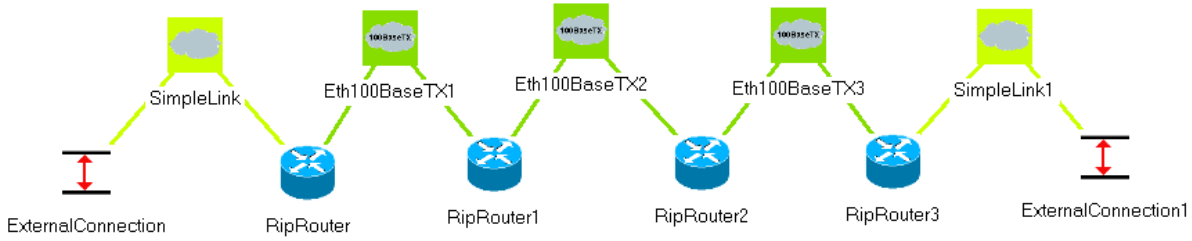


Figure 18: Virtual network for throughput test design

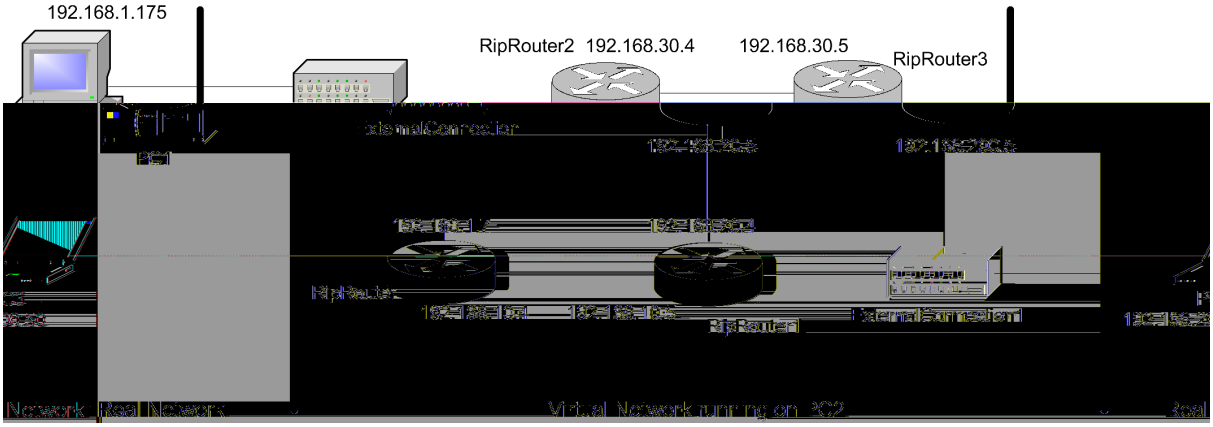


Figure 19: Throughput test network structure

A very large file has been transferred from PC1 to PC3. As Figure 20 demonstrates, the speed was usually above 3,000 kilobytes per second, which is almost 25 Mbps. The machine running NetSim had CPU usage about 50-60% during the transfer.

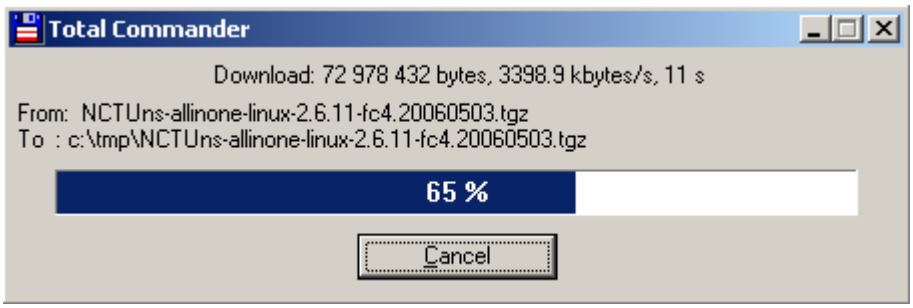


Figure 20: Data transfer speed over the virtual network

Running a Real Application on the Virtual Node

The final test is focused on the Remote Sockets feature. It uses an application that is known to work with current experimental version of Remote Sockets – WinSCP. The test configuration is shown in Figure 21.

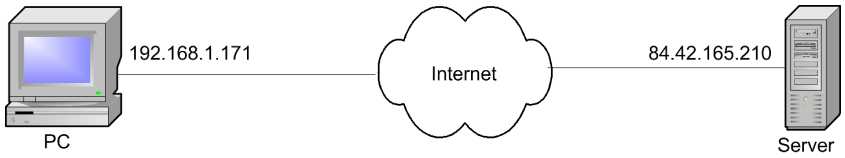


Figure 21: Remote Sockets test

The PC is running virtual network; again, the example network provided with NetSim is used (see Figure 15 on page 43). The WinSCP application is redirected to the "Workstation2" node, connected to the remote server in the Internet, and large files are downloaded and uploaded.

The file transfer worked well; large file of about 10 MB was uploaded and then downloaded. The screenshot of packet capture at the same interface where the virtual network is connected is shown in Figure 22. It evidences that the file transfer was really from "Workstation2" node (192.168.65.50).

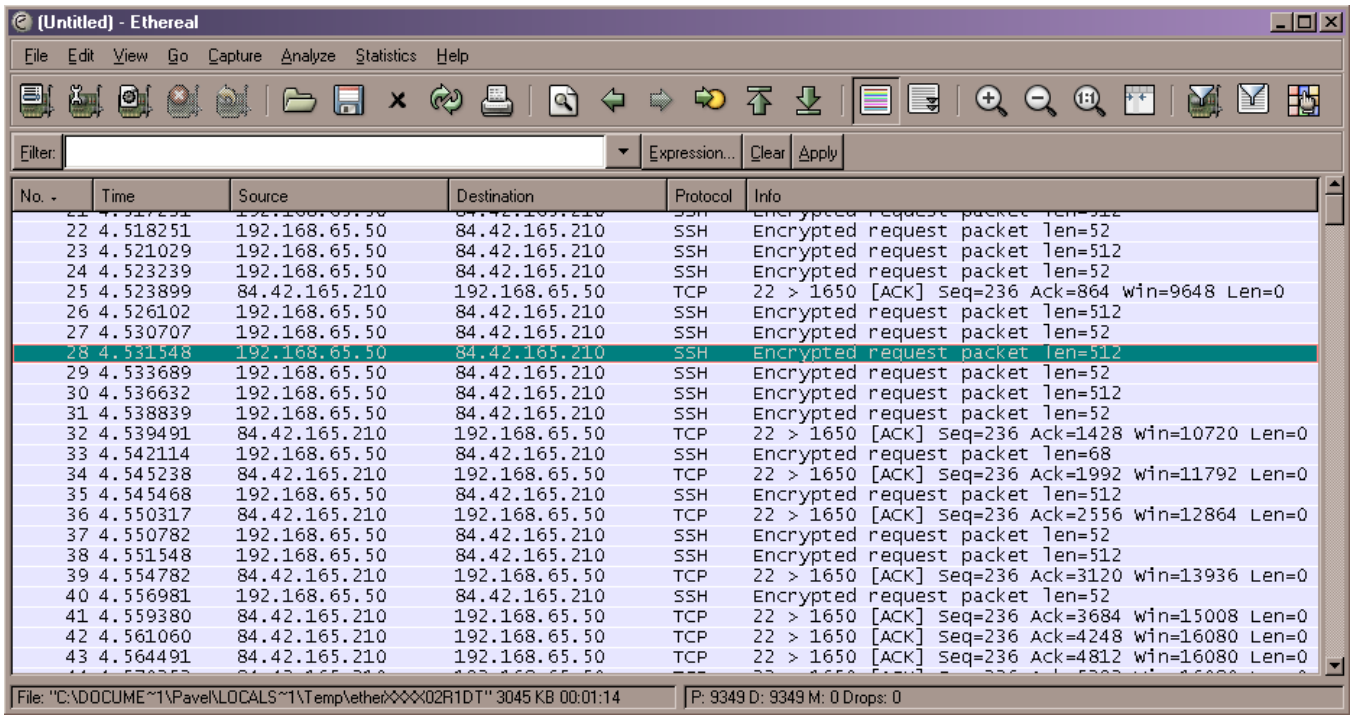


Figure 22: Ethereal screenshot while uploading the file

6. Conclusions

The purpose of this chapter is to summarize the work, the fulfillment of its goals and contributions. It also suggests possible directions of further work.

Summary

According to an original thesis submission, a main goal of this work should be providing an implementation of a computer network simulator that will be suitable for teaching, testing of network monitoring tools, and deploying so called ‘honeypots’. The first chapter discusses the sense of such usage combination, shows that some requirements for teaching versus honeypots are contradictory and that it would be better to provide just a tool for teaching and testing. In addition, existing simulators are compared and their feasibility for such usage is dissected.

The second chapter introduces the NetSim architecture, discusses the alternatives and reasons for the choices made. Chapter 3 then explains in detail how the ideas from Chapter 2 are implemented.

Chapter 4 introduces the Remote Sockets feature that allows redirection of any network communication to a node in the virtual network; this unique feature is not supported by any other simulator.

Chapter 5 demonstrates some usage scenarios and NetSim performance under various test conditions.

Comparison to Other Simulators

A comparison according to some impartial and measurable criteria would be a very hard task: the main purpose of each existing simulator is different; therefore, formulation of criteria to compare is almost impossible. For example, a speed of the simulation could be a well measurable criterion. However, a lower speed is not a handicap for teaching and in most cases also testing.

Discussion about features that are available in individual simulators is presented in the first chapter. A table summarizing the features of existing simulators is shown on page 12; the same table having one additional column for NetSim follows.

Requirement	*NS-2*	*CNET*	*JNS*	*OPNET*	*AdventNet*	*NCTUns*	*NetSim*
Easy installation	✗	✗	✓	✓	✓	✗	✓
Requires writing no code to build simulated network	✗	✗	✗	✓	✓	✓	✓
Simulation at link layer level	✗	✓	✗	✓	✗	✓	✓
Network structure visualization	●	✗	✗	✓	✓	✓	✓
Data flow visualization	✓	✗	●	✓	✗	✓	✗
Extensibility of network components	✓	✗	●	?	✗	✗	✓
Extensibility of supported protocols	✓	✓	●	✓	✗	✓	✓
Wide range of protocols supported	✓	✗	✗	✓	✓	✓	✗
Interconnection to the real network	✓	✗	✗	?	✗	✓	✓
Runs on Linux	✓	✓	✓	✗	✓	✓	✓
Runs on Windows	✗	✗	✓	✓	✓	✗	✓
Source code available	✓	✓	✓	✗	✗	✓	✓

✓ present ● not completely present

✗ not present ? not known

The table shows that NetSim meets most requirements among of all simulators. Moreover, the missing support for wide range of protocols cannot be simply compared to other simulators since this is just the first version. In addition, data flow visualization can also be implemented in some of future releases.

It is obvious that the support of some feature by NetSim does not mean that the feature is the best. For example, other existing simulators have much more sophisticated graphic environment. However, those drawbacks would certainly be gradually eliminated in new versions.

The next chapter describes separate requirements in more detail.

Goals Fulfillments

The features that should be included in NetSim are listed in Chapter 1 (see Project Goals Revisited, page 13). The following table summarizes them and provides comments to the implementation.

Requirement	*Fulfilled?*	*Comment*
Easy installation	Yes	The installation can be done just by copying the directory tree to the target machine; on Windows is then finished by executing one batch script. (For Remote Sockets usage the installer is currently needed; however, it asks only a few simple questions.)
No code when creating the simulated network	Yes	A graphic application for designing the virtual network structure is implemented; just modifying configuration files of separate network components is needed. On Linux, where the graphic tool is not currently available, creating manually one additional configuration file is required.
Link layer simulation	Yes	The simulation is done at a link layer, Ethernet implementation is provided.
Network structure and data flow visualization	Partially	Network structure visualization is satisfied thanks to the graphic application, which allows also modification of the network. Data flow visualization is not implemented in this version.
Extensibility	Yes	The simulator allows the addition of new components and protocols. It is done by implementing a class (in any language for .NET) that derives from an abstract class provided or implements a specific interface. Moreover, a set of predefined components in the graphic application can also be easily extended by modifying application configuration file.
Wide range of protocols	No	Only TCP/IP protocol suite and RIPv2 for routing are implemented. Therefore, the usability for testing of network monitoring programs is limited (especially due to missing SNMP).
Multiplatform solution	Partially	NetSim is implemented over the .NET Framework, which is currently available for Windows and Linux. It will be hopefully implemented for some other platforms in the future.

As stated in the table, almost all requirements are satisfied. The only one is completely missing due to a lack of libraries of protocol implementations – it is excusable since this is the first version of the application. Data flow visualization is also missing because there was no time for implementation; it will be certainly included in a new version. Finally, having the implementation for Windows and Linux only is not so crucial since those are the two leading platforms over the world.

Further Work

Since this is the first version of NetSim, there are many things to improve. Obviously, first, unfinished and not well-tested parts of the simulator should be completed. Afterwards new features and libraries can be added. A list of things to do follows – current priority corresponds to the list order. However, priorities are subject to change depending on the users feedback and the purpose for which the simulator will be mainly used.

1) Complete *Remote Sockets* feature. Windows Sockets Layered Service Provider interface is not completely implemented; there are obviously many programs not working. Moreover, the installation of this LSP breaks the network connection on many computers. This feature should be well tested on all versions of MS Windows.

   Running *Remote Sockets* over the network (via TCP channel) results currently in non-trivial problems; only local redirection via shared memory can be used for now. Solving this issue will help improve the simulation performance because applications would be able to run on the different machine than the simulator.

   LSP should be also rebuilt using new release of Platform SDK (R2) from March 2006, currently the release SP1 from May 2005 is used.

2) Exclude SharpPcap library from the project. SharpPcap is needed for communication with the real network. However, this library is capable of crating its own internal representation of packets etc. NetSim needs just capture an array of bytes and send raw array of bytes to the network, nothing else. Therefore, the library was modified and currently the main part of it is not needed. Own managed interface to WinPcap (libpcap) would be a much more plain solution.

3) Deeply test the simulator in a multiprocessor environment for race conditions. Because the author does not have access to any multiprocessor machine, the simulator was not tested in this environment. Although it was implemented carefully with respect to such conditions, because threads are widely used in the core of the simulator, it is possible that some bad situations are not handled.

4) Implement SNMP and DHCP support.

5) Add capturing of packets support; at least dump packets traveling over the link to a file that can be read by third-party tools such as Ethereal [5].

6) Implement data flow visualization in a graphic application.

7) Move all strings displayed to the user to separate file that can be localized.

8) Implement more network protocols (OSPF, IPv6).

9) Implement more network components (Application running FTP, …; another link layer protocols such as ATM or FDDI).

10) Add console interface for remote sockets. This will be useful to redirect applications in the script.

11) Add Remote Sockets for Linux.

7. References

[1] AdventNet, Inc.: AdventNet Simulation Toolkit. http://www.adventnet.com/products/simulator/ 10

[2] BenDi, PriorityQueue. http://www.codeproject.com/csharp/PriorityQueue.asp 19

[3] Clifton, Mark: .NET's ThreadPool class – Behind the Scenes. http://www.codeproject.com/csharp/threadtests.asp 19

[4] Comer, Douglas E. (2006): Internetworking with TCP/IP, fifth edition. Pearson Prentice Hall, Upper Saddle River, New Jersey. 31

[5] Ethereal Network Analyzer. http://www.ethereal.com/ 52

[6] Gal, Tamir: SharpPcapLibrary, http://www.tamirgal.com/home/dev.aspx?Item=SharpPcap 22, 59

[7] Keshav, Srinivasan: Computer Science Department Technical Report 88/472, UC Berkeley, 1988. http://www.cs.cornell.edu/skeshav/real/ 11

[8] Krasser, Sven at al.: The use of honeynets to increase computer network security and user awareness. http://www.ece.gatech.edu/research/labs/nsa/papers/use_of_honeynets.pdf 6

[9] McDonald, Chris: CNET Network Simulator. http://www.csse.uwa.edu.au/cnet/ 9

[10] MSDN Library, NDIS Drivers. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/NetXP_d/hh/NetXp_d/102gen_24174df5-78af-48f3-8853-563c44c2e852.xml.asp 35

[11] NS-2 Network Simulator. http://www.isi.edu/nsnam/ns/ 9

[12] OPNET Technologies: OPNET IT Guru Academic Edition. http://www.opnet.com/ 10

[13] Prošek, Ladislav (2006): Shared Memory Channel, Phalanger Project, http://www.php-compiler.net 59

[14] Scalable Network Technologies: QualNet Network Simulator. http://www.scalable-networks.com/ 11

[15] Toub, Stephen: ManagedThreadPool. http://www.gotdotnet.com/Community/UserSamples/Details.aspx?SampleGuid=BF59C98E-D708-4F8E-9795-8BAE1825C3B6 19, 56

[16] Vollset, Einar W.: Java Network Simulator. http://jns.sourceforge.net/ 10

[17] Wang, S.Y. et al.: The Design and Implementation of the NCTUns 1.0 Network Simulator, Computer Networks, Vol. 42, Issue 2, June 2003, pp.175-197. 10

[18] Windows Sockets 2 Service Provider Interface documentation. ftp://ftp.microsoft.com/bussys/winsock/winsock2/WSSPI22.DOC 36

8. Appendixes

Appendix A – other network simulators

NS-2

http://www.isi.edu/nsnam/ns/

NS-2 is one of the first network simulators which was funded by DARPA. It is written in C++, the simulation requires writing scripts in OTcl (object extension of Tcl language). After about 15 years of development, it is stable and many contributors created extensions that support other protocols than TCP/IP. Requires Unix-like operating system, the source code is available.

QualNet

http://www.scalable-networks.com/

A commercial network simulator that supports a wide range of protocols; some of them are also written by third parties. Even though it is commercial, they provide the simulator for research and education to the universities. Runs on Windows, Unix-like OS, Solaris, and Apple. Core written in C/C++, some tools in Java. Includes GUI for creating virtual network topology.

CNET

http://www.csse.uwa.edu.au/cnet/

Developed for education purposes at The University of Western Australia. Written in C, it requires writing scripts describing the simulated environment. Requires Unix-like operating system. The simulation can be affected through C API.

AdventNet Simulation Toolkit

http://www.adventnet.com/

A commercial simulator running on Windows, Linux and Solaris. Supports SNMP, Cisco IOS, it has GUI for creating simulated network topology.

REAL

http://www.cs.cornell.edu/skeshav/real/

Developed at Cornell University for research of flow and congestion control. Written in C, it requires Unix-like OS, Solaris, or some others. Currently also GUI RealEdit for building network topology is available.

RouterSim

http://www.routersim.com/

Simulator intended primarily for training of Cisco IOS, provides experience to pass Cisco exams. Only Windows platform is supported.

NIST ATM Network Simulator

http://w3.antd.nist.gov/Hsntg/prd_atm-sim.html

Implemented at National Institute of Standards and Technology; written in C. As the name shows, it is ATM only simulator. Requires Unix-like operating system.

Enhanced Network Simulator

http://www.cse.iitk.ac.in/~bhaskar/tens/

Developed at Indian Institute of Technology, it is an extension of NS-2 simulator. Adds some features not included in the original NS-2 like mobility support.

NAB

http://nab.epfl.ch/

Written in Objective Caml, it is targeted at wireless ad hoc networks.

JNS

http://jns.sourceforge.net/

Java Network Simulator is Java implementation of NS-2. It is not so complete as the original, but simplifies creating scripts and provides the same output as NS-2, so it can be processed by any tool written for NS-2.

NCTUns Network Simulator and Emulator

http://nsl10.csie.nctu.edu.tw/

Developed at National Chiao Tung University by Prof. S. Y. Wang and his students. Supports both wired and wireless networks, mobility and many protocols, simulations can run on a remote computer. Requires Linux, actually Fedora Core 4.

OPNET

http://www.opnet.com/

A commercial network simulator; the subtle graphic interface allows to model not only traditional wired networks, but also wireless networks and simulate node mobility. Some licenses for universities for education and research are available.

Appendix B – Source Code Description

Core project

Core simulator functionality – thread pool for executing network events, server-side for remote sockets, abstract classes for network structure implementation.

- **Collections**
  - *ByteArrayList.cs* – Collection used for storing bytes of packets.
  - *DoubleHashtable.cs* – Hashtable hashing both keys to values and values to keys.
  - *PriorityQueue.cs* – Priority queue; implemented as binary heap.
  - *SortedList2.cs* – Sorted list allowing duplicate entries.
- **Configuration**
  - *NetSimConfig.xml* – Virtual network configuration file.
  - *NetSimConfig.xsd* – XSD schema for checking network configuration files.
- **Structure**
  - *Adapter.cs* – Abstract class for creating adapters; SimpleAdapter implementation.
  - *Application.cs* – Abstract class for creating applications running on the virtual nodes.
  - *Interfaces.cs* – Some interfaces used in the Core project.
  - *Link.cs* – Abstract class for creating virtual network links; SimpleLink implementation.
  - *Module.cs* – Abstract class for creating modules.
  - *NetworkAddress.cs* – Basic class for addresses used over the network.
  - *Node.cs* – Virtual node that carries adapters, modules, and applications.
  - *Packets.cs* – Abstract class for crating packets, packets for carrying known and unknown data.
  - *ProtocolSettings.cs* – Ancestor for arbitrary protocol settings associated with adapters.
  - *StructureObject.cs* – Common ancestor for all objects in the virtual network.
- *AssemblyResources.cs* – Access to assembly resources (localized strings etc.).
- *EventLog.cs* – Log for errors, warnings and notices.
- *Exceptions.cs* – Exceptions used in the simulator.
- *Factories.cs* – Abstract class for creating packet factories.
- *ManagedThreadPool.cs* – Thread pool of threads waiting for the network event to execute. Based on Stephen Toub's ManagedThreadPool [15]
- *NetSimSocket.cs* – Socket at the virtual node; implements the same interface as .NET Framework Socket class.
- *RemoteControl.cs* – An object that can be referenced remotely to control the simulation.
- *RemoteSockets.cs* – Controls sockets on the virtual network created remotely. Can be accessed remotely to execute socket operations.

- *SimulationTimer.cs* – Timer used by the simulator.
- *Simulator.cs* – The simulator – initializes and runs the simulation.
- *Strings.resx* – Resource strings that can be localized.

Ethernet project

As a specific part, Ethernet functionality was implemented independently from the main simulator library to be an example project for creating another libraries.

- **Modules**
  - *LearningSwitch.cs* – Simple non-configurable learning switch.
- *ArpCache.cs* – ARP cache for Ethernet adapter.
- *EthFrame.cs* – Ethernet packet implementation.
- *EthMacAddress.cs* – Ethernet address.
- *EthernetAdapter.cs* – Adapter for Ethernet links family.
- *EthernetLink.cs* – 10Base2 and 100BaseTX Ethernet link implementation.

Gui project

Graphic application that simplifies virtual network creation.

- **Data**
  - *Exceptions.cs* – Exceptions used in GUI.
  - *GeneralData.cs* – Graphic application configuration parsing and handling.
  - *GuiProjectData.cs* – Project-specific configuration used in graphic application parsing and handling.
  - *ProjectData.cs* – Virtual network configuration files processing.
  - *XmlData.cs* – Abstract class for XML files processing.
- **Forms**
  - *AboutForm.cs* – About screen.
  - *ErrorForm.cs* – Form used when an unexpected error occurs.
  - *ItemAppearanceForm.cs* – Changing the appearance of an graphic item in the virtual network.
  - *MainForm.cs* – Main application form.
  - *NewVersionForm.cs* – Form with current and newly available version information.
  - *NodeItemsListForm.cs* – Form used for displaying the list of items that can be added to the node.
  - *NodeProperties.cs* – Properties of the virtual network node.
  - *RenameItem.cs* – Form used for renaming various network items.
  - *SplashScreen.cs* – Splash screen displayed while the application is starting.
- *App.ico* – Application icon.
- *Console.cs* – Console within the graphic application.
- *FileSystem.cs* – Helper methods related to filesystem.
- *GuiConfig.xml* – Graphic application configuration file.
- *SimulatorControl.cs* – Communication with the simulator to control the simulation process.

Installer project

Microsoft Windows Installer project that builds MSI package for installation.

LSP project

Layered Service Provider installed to the system; provides redirection of socket function calls. Loads LSPWrapper to communicate with the simulator via .NET Remoting. Based on LSP example provided by Microsoft in Platform SDK.

Only files in ***bold*** were modified or newly added, others were left unchanged or only some minor modifications were made.

- **common**
  - *provider.cpp* – Common support functions for enumerating WinSock catalog.
- **install**
  - *instlsp.cpp* – Installer for inserting the LSP to the catalog.
  - *lspadd.cpp* – Installing provider.
  - *lspdel.cpp* – Removing provider.
  - *lspmap.cpp* – Handling provider dependencies.
  - *lsputil.cpp* – Helper functions used by other parts of the installer.
  - *prnpinfo.cpp* – Printing information about the provider.
- **netsim**
  - ***netsimspecific.cpp*** – Routines for initializing provider and handling information about which processes are redirected.
  - ***netsimspi.cpp*** – WinSock interface functions called by the LSP that interacts with the simulator internally.
- **nonifslsp**
  - *asyncselect.cpp* – Hidden window for interceptiong WSPAsyncSelect calls.
  - *extension.cpp* – WinSock extension functions intercepting.
  - *lspguid.cpp* – GUID for provider catalog entry.
  - *overlap.cpp* – Overlapped I/O operations handling.
  - *sockinfo.cpp* – Mapping between upper and lower layer sockets.
  - ***spi.cpp*** – Service provider interface functions called by the upper layer.

LSPWrapper project

Managed part of NetSim LSP; loaded by the unmanaged LSP. Handles communication with the simulator via .NET Remoting.

- *LSPWrapper.cpp*– Initializes remoting, stores remote references to simulator objects.
- *init.cpp* – Exported functions called by unmanaged LSP to initialize DLL.
- *spi.cpp* – WinSock LSP functions called by LSP when appropriate function is called by the application that is being redirected.

LSPWrapperControl project

Graphic application that controls LSP to redirect socket communication of selected applications.

- *App.ico* – Application icon.
- *MainForm.cs* – Application logic + user interface.

Library project

Basic library of network structure objects.

- **Applications**
  - *RIP.cs* – Routing Information Protocol (version 2).
  - *SimpleWebServer.cs* – Simple web serever that can reply to web requests with page or error according to configuration file.
- **Modules**
  - *ExternalConnection.cs* – Connection to the real network.
  - *IP.cs* – IP (currently version 4 only) protocol implementation.
  - *IcmpModule.cs* – ICMP protocol.
  - *Repeater.cs* – Module that sends immediately everything that receives.
  - *SocketModule.cs* – Module required for socket applications or remote sockets.
  - *TCP.cs* – TCP protocol implementation.
  - *UDP.cs* – UDP protocol implementation.
- *Addresses.cs* – IP address.
- *Packets.cs* – IP, ICMP, TCP, UDP, ARP, and SocketDataPacket packets and their factory. All those packets are used by applications and modules in the library.
- *ProtocolSettings.cs* – IP settings that are associated with an adapter.

NetSimConsole project

Console application that runs the simulator.

- *App.ico* – Application icon.
- *NetSimConsole.cs* – Console application; parsing and executing user commands.

SharpPcap project

SharpPcap library. The main part of the library is currently not used (see Further Work, page 52) [6].

ShmChannel project

Shared memory channel used for remoting communication between processes [13].

Appendix C – User's Manual

Building the source code

To build the source code, following components should be installed:

- .NET Framework 1.1 SDK
  (Usually installed together with the Visual Studio 2003; otherwise available at Microsoft's download website.)
- Visual Studio 2003
- Windows Server 2003 SP1 Platform SDK
  This is not the most recent version of the SDK; however, it is the one used for the simulator development. Upgrading to the newer release is in the TODO list. It can be also downloaded at http://www.microsoft.com/downloads/.
  The only required component of Platform SDK is "MS Windows Core SDK."

Then, the *NetSim-1.0-alpha.src.zip* from the NetSim package should be unpacked to the destination directory and *NetSim.sln*, which can be found in the *NetSim* directory, opened in the Visual Studio. Next, one of four configurations can be selected: Debug, Release, MonoDebug, or MonoRelease. The first two of them are intended for Windows, the others for Mono running on Linux.

After building any of the pre-set configurations[21], the *Deployment* directory will contain executable files and libraries that can be copied to the target machine. In addition, in the sub-directory *NetSim/Installer* an installer package will be created for Windows configurations.

Installation

Prerequisities

NetSim requires some components installed on your computer.

- **.NET Framework v1.1**

  This is a required component. For Windows, go to http://www.microsoft.com/downloads/details.aspx?FamilyID=262d25e3-f589-4842-8157-034d1e7cf3a3&DisplayLang=en
  (or visit http://www.microsoft.com/downloads/ and search for *.NET Framework 1.1 Redistributable Package*).

  For Linux, install Mono. Go to http://go-mono.com/sources-stable/, download the latest release of version 1.1 and install.

[21] One more step is necessary for source package downloaded from the NetSim website. For security reasons, key pairs for signing assemblies that are required to have a strong name are excluded. Each developer should create his/her own: NetSim/NetSim.snk and NetSim/ShmChannel/ShmChannel.snk. This task is necessary only when using the downloadable package; the source code tree on a CD attached to this work includes those files.

- **Packet capture library**

  This component is not required if connecting virtual network with the real network will not be used. However, it is recommended to prepare the environment for this feature.

  For Windows, download WinPcap. At http://www.winpcap.org/install/ select the latest stable version (currently 3.1) and install.

  For Linux, LibPcap library might be already installed (it is included in many distributions). If not, download LibPcap at http://sourceforge.net/projects/libpcap/ and install.

Two types of installation are available. One is very easy – just decompressing file into the target directory; however, remote sockets feature is not available if you use this one. Another one is regular Microsoft installation package. (Of course, available only on Windows, nevertheless, remote sockets are windows-based and not available on Linux.)

Installation type 1

Decompress the file *NetSim-1.0-alpha.zip* (*NetSim-1.0-alpha.tar.gz* for Linux) into the destination directory. On Windows, go to *Tools* directory and execute *gacregister.bat*, which will register some assemblies to the Global Assembly Cache. Such operation is not required for Linux, because no assembly in GAC is needed for command-line simulator interface.

Network simulator console is located in *Bin* directory (*netsim.exe*), for launching simulator graphic interface use either *NetSim.cmd* in *Gui* directory or launch *Bin/NetSimGui.exe* specifying GUI main configuration file as the first parameter (*Gui/GuiConfig.xml*).

Installation type 2

This type of installation is available only for Windows and is required if you want to use remote sockets. Run *Setup.exe* after unpacking *NetSim-1.0-alpha.install.zip* and follow the setup instructions.

Uninstallation

For Windows: If the NetSim was installed through the installation wizard, either run it again or go to Control Panel → Add or remove programs, select “Network Simulator” and click Remove.

In all other cases, deletion of the directory where the simulator files were copied or unpacked is sufficient.

Using the command line

The command line executable named *netsim.exe* can be found in *Bin* directory. It requires the path to virtual network configuration file as a command line parameter and accepts some switches. The usage is following (individual command line switches are described in the table):

```
netsim [-rc:shm] [-rs:shm] [-pt:simple|adv|none] [-h] <configuration file>
```

SWITCH	ACCEPTED VALUES	DESCRIPTION
`-rc`	`shm`	Turns on Remote Control, which allows programmatic access to the simulator. The simulator will provide remotely accessed object that can be used for controlling the simulation from another process via .NET Remoting. The value represents a channel that can be used for accessing the object, currently only `shm` is available. (`shm` = ShmChannel; communication through the shared memory)
`-rs`	`shm`	Turns on Remote Sockets, a feature that enables a redirection of network communication of any application. (`shm` = Communication via shared memory) Not available for Linux.

`-pt`	`simple`
	`adv`
	`none`
		Type of prompt displayed.
		
		`simple` – The simulator will be configured and started and only stopping will be possible
		
		`adv` – Prompt enabling the user to control the simulation, enable/disable network components and exit the simulation will be available. This is the default value.
		
		`none` – No user interaction. The simulator will be configured and started and the console will just wait for simulator shutdown. Useful with `-rc` only.

`-h`	`---`	Displays help for the usage.

Using graphic interface

The GUI application contains three main panels. The largest one is used for virtual network design and is available only if a project is opened. On the left, there is a panel containing predefined types of nodes and links that can be added to the network by dragging the icon to the previously described panel. The last one is placed at the bottom: it displays error and information messages, and on other two tabs (while the simulator is running) simulator console and events written to the simulator log. An application screenshot is shown in Figure 23.

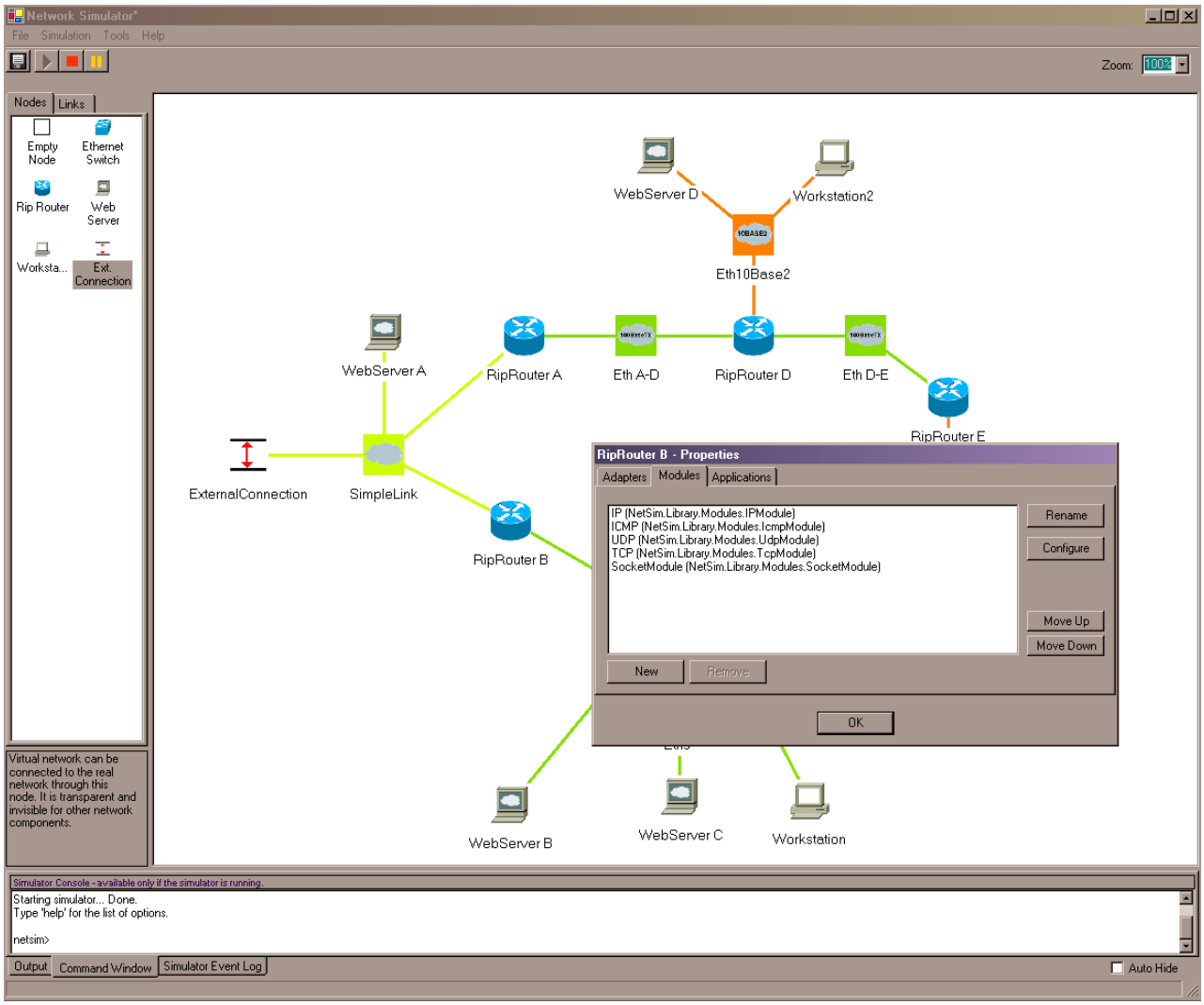


Figure 23: Graphic application screenshot

Small “HOWTO“ list of common actions follows.

Creating new project

Go to File → New… and choose the directory. Enter the new project name; the directory of that name and a file with the same name and .xml extension inside will be created. That .xml file is the main configuration file for the virtual network and should be selected in the future while opening the project.

Adding new link to the network

On the panel on the left, select ‘Links’ tab. Then use Drag&Drop to move the appropriate icon to the network design on the right. Right-click the new icon to view a context menu. Dialog boxes for renaming or changing the link color can be shown, the link deleted or, if the link has some configurable properties, its configuration file can be opened for editation.

Adding new node to the network

On the panel on the left, select the ‘Nodes’ tab and add the appropriate node to the network design exactly as described above for the link. In addition to the link, the ‘Properties’ item is

available in the context menu. Its activation will show a dialog box that allows choosing adapters, modules, and applications for the node.

Configuring node and connecting the node to the link

Right-click the node and choose ‘Properties’. The dialog box with three tab pages will appear. On the first tab, adapters associated with the node can be added, removed, and configured. To add a new adapter, click ‘New’ and select the adapter type. To remove, select an existing adapter and click ‘Remove’. If the adapter has some configurable properties, double-click the adapter on the list or click ‘Configure’ to open its configuration file. Finally, to connect the adapter to the link, select the adapter and choose the link name in the combo box below.

The procedure for Modules and Applications is almost identical, except that for modules the order in which they are listed can be altered. The first module in the list will be the first that will get the packet received by any adapter.

Starting the simulation

When the virtual network is configured, the simulation can be started by choosing Simulation → Start or clicking the Start button on the toolbar. The application will start the simulator in a separate process and connect to it to control and monitor the simulation process. If there was some error, for example in the configuration, the simulator would exit and an error message can be found either in ‘Command Window’ or ‘Output’ window.

Enabling/disabling network component

While the simulator is running, it is possible to enable/disable the link or node, which can simulate the node or cable failure. Right click to the node/link and choose Enable/disable, it will toggle the element state. Disabled elements are displayed with a red background of the title.

Using the simulator command line

It is possible to control the simulator via either GUI, or the command line. To enter a command, select ‘Command Window’ on the bottom and click inside. Move the cursor to the end (or try to type, the first key press will move the cursor), write the command and press enter. The simulator reply will be displayed in the same window. Do not be confused by the presence of commands you have not written, some commands sent by GUI are sent via the command line and they are displayed here. To get a list of available commands use ‘help’.

Connecting the virtual and live network

A special module called ExternalConnection is used for interconnection between real and virtual network. Add node called Ext. Connection to the network, it includes such module. Second, add Simple link and connect the previously added node to it. In addition, connect all nodes that should be visible from the real network (and that will get packets that will be captured from the real network) to the Simple link. Finally, open properties of new ExternalConnection node and open configuration file of ExternalConnection module. In the configuration file, specify the name of a local computer’s real interface that will be used for capturing and sending packets. If you do not know the name, just run the simulator and it will list all possible interface names.

Changing the MAC and IP address of a node interface

If using IP protocol, generally after adding a node to the virtual network, a change of IP address and physical interface address is needed. Open node properties, go to 'Adapters' tab page, the appropriate adapter and click 'Configure'. Change the IP and physical address appropriately in an opened configuration file.

Changing the editor for configuration files

If the default 'notepad.exe' is not enough, open *GuiConfig.xml*, which is located in *Gui* subdirectory in the installation directory. Find the 'ExternalEditor' XML tag and change the notepad to the path to the executable file of your favorite editor. Restart GUI application if running.

Adding a new node type to the list of predefined nodes

Create a new directory under *Gui/Templates/Nodes*. In that directory, create a file *Config.xml* that will contain the only element 'Node' as a root element. Write node configuration (including adapters, modules, and applications) in this file. The contents of this file will be copied to the main configuration file of the virtual network while adding a new node. If some items require their own configuration files, create them in this directory or subdirectories and refer to them by relative path (relative to the *Gui* directory). It would be the best to follow the pattern used by other predefined nodes, that is to create three subdirectories next to the *Config.xml*: *Adapters*, *Modules*, *Applications* and place the appropriate configuration files into that subdirectories.

The procedure for creation of a predefined Link, Adapter, Module, or Application is almost the same; take the existing entries in the *Templates* directory as an example.

Example virtual network

A part of the main configuration file for an example virtual network, which is distributed with NetSim, follows. That network is also displayed on GUI screenshot in Figure 23. Instead of describing it generally here, XML comments are presented in the code. The diagram of the network is shown in Figure 24.

```
<?xml version="1.0" encoding="utf-8"?>

<!-- XML configuration file is validated against the XML schema -->
<NetSimConfig xmlns=http://pavelnovak.eu/NetSimConfig
              xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
              xsi:schemaLocation="http://pavelnovak.eu/NetSimConfig NetSimConfig.xsd">

  <!-- Assemblies to load and search for classes. Paths are absolute or relative to
       the directory where Simulator executable file resides. -->
  <Extension path="../../../Bin/NetSimEthernet.dll">
    <!-- Packet factory is used to create internal structure of packets that
         are newly added by this extension library. More than one packet factory
         can be listed here. -->
    <PacketFactory class="NetSim.Ethernet.EthernetPacketFactory" />
  </Extension>
  <Extension path="../../../Bin/NetSimLibrary.dll">
    <PacketFactory class="NetSim.Library.LibPacketFactory" />
  </Extension>

  <!-- Network structure -->
  <Network>
```

```
<!-- List of nodes in the virtual network -->
<Nodes>

  <!-- Node providing the interconnection between virtual and real networks -->
  <Node name="ExternalConnection" class="NetSim.Core.Node">
    <!-- Simple adapter does not add any link layer headers -->
    <Adapter name="SimpleAdapter" link="SimpleLink"
                                        class="NetSim.Core.SimpleAdapter" />
    <!-- Module capturing the real network traffic and sending virtual
         network packets to the real network -->
    <Module name="ExtConnection" class="NetSim.Library.Modules.ExternalConnection
            config="Nodes/ExternalConnection/Modules/
                                    ExternalConnection/ExternalConnection.xml" />
  </Node>

  <!-- Node with a web server that replies by simple page to browser requests -->
  <Node name="WebServer A" class="NetSim.Core.Node">
    <!-- Adapter for ethernet-like links; here is connected to the SimpleLink,
         however, we expect that the real network is Ethernet. -->
    <Adapter name="Ethernet" link="SimpleLink" config="Nodes/WebServer
       class="NetSim.Ethernet.EthernetAdapter" A/Adapters/Ethernet/Ethernet.xml" />
    <!-- Module implementing IP protocol -->
    <Module class="NetSim.Library.Modules.IPModule" name="IP"
                                   config="Nodes/WebServer A/Modules/IP/IP.xml" />
    <!-- Module implementing ICMP protocol. Cooperates with IP. -->
    <Module class="NetSim.Library.Modules.IcmpModule" name="ICMP" />
    <!-- TCP and UDP protocol modules, require IP, indeed -->
    <Module class="NetSim.Library.Modules.UdpModule" name="UDP" />
    <Module class="NetSim.Library.Modules.TcpModule" name="TCP" />
    <!-- This module provides support for NetSimSockets -->
    <Module class="NetSim.Library.Modules.SocketModule" name="SocketModule" />
    <!-- Finally the only one application on this node -->
    <Application class="NetSim.Library.Applications.SimpleWebServer"
        name="SimpleWebServer" config="Nodes/WebServer A/Applications/
                                          SimpleWebServer/SimpleWebServer.xml" />
  </Node>

  <!-- Node with more than one adapter to route packets -->
  <Node name="RipRouter A" class="NetSim.Core.Node">
    <Adapter name="Ethernet" link="SimpleLink" class="NetSim.Ethernet.EthernetAdapter"
                        config="Nodes/RipRouter A/Adapters/Ethernet/Ethernet.xml" />
    <Adapter name="Ethernet1" link="Eth A-D" class="NetSim.Ethernet.EthernetAdapter"
                         config="Nodes/RipRouter A/Adapters/Ethernet1/Ethernet.xml">
    <!-- IP module is capable of routing -->
    <Module class="NetSim.Library.Modules.IPModule" name="IP"
                                    config="Nodes/RipRouter A/Modules/IP/IP.xml" />
    <Module class="NetSim.Library.Modules.IcmpModule" name="ICMP" />
    <Module class="NetSim.Library.Modules.UdpModule" name="UDP" />
    <Module class="NetSim.Library.Modules.TcpModule" name="TCP" />
    <Module class="NetSim.Library.Modules.SocketModule" name="SocketModule" />
    <!-- RIP application receives and sends RIP update packets and modifies
         routing table located in IP module -->
    <Application class="NetSim.Library.Applications.Rip" name="RIP"
                             config="Nodes/RipRouter A/Applications/Rip/Rip.xml" />
  </Node>

  <!-- Node with many adapters that acts as Ethernet switch. Some of the
       adapters are not connected. -->
  <Node name="EthSwitch" class="NetSim.Core.Node">
    <Adapter name="Ethernet" link="Eth" class="NetSim.Ethernet.EthernetAdapter"
                         config="Nodes/EthSwitch/Adapters/Ethernet/Ethernet.xml" />
    <Adapter name="Ethernet1" link="Eth1" class="NetSim.Ethernet.EthernetAdapter"
                        config="Nodes/EthSwitch/Adapters/Ethernet1/Ethernet.xml" />
    <Adapter name="Ethernet2" link="Eth2" class="NetSim.Ethernet.EthernetAdapter"
                        config="Nodes/EthSwitch/Adapters/Ethernet2/Ethernet.xml" />
    <Adapter name="Ethernet3" link="Eth3" class="NetSim.Ethernet.EthernetAdapter"
                        config="Nodes/EthSwitch/Adapters/Ethernet3/Ethernet.xml" />
```

```
        <Adapter name="Ethernet4" link="Eth4" class="NetSim.Ethernet.EthernetAdapter"
                            config="Nodes/EthSwitch/Adapters/Ethernet4/Ethernet.xml" />
        <Adapter name="Ethernet5" link="" class="NetSim.Ethernet.EthernetAdapter"
                           config="Nodes/EthSwitch/Adapters/Ethernet5/Ethernet.xml" />
        <Adapter name="Ethernet6" link="" class="NetSim.Ethernet.EthernetAdapter"
                           config="Nodes/EthSwitch/Adapters/Ethernet6/Ethernet.xml" />
        <Adapter name="Ethernet7" link="" class="NetSim.Ethernet.EthernetAdapter"
                           config="Nodes/EthSwitch/Adapters/Ethernet7/Ethernet.xml" />
        <Adapter name="Ethernet8" link="" class="NetSim.Ethernet.EthernetAdapter"
                           config="Nodes/EthSwitch/Adapters/Ethernet8/Ethernet.xml" />

        <!-- The only one module providing the learning switch logic -->
        <Module class="NetSim.Ethernet.LearningSwitch" name="EthLearningSwitch" />
      </Node>

      <!-- Node with no application that is ready to accept application traffic
           via Remote Sockets -->
      <Node name="Workstation" class="NetSim.Core.Node" >
        <Adapter name="Ethernet" link="Eth4" class="NetSim.Ethernet.EthernetAdapter"
                           config="Nodes/Workstation/Adapters/Ethernet/Ethernet.xml" />
        <Module class="NetSim.Library.Modules.IPModule" name="IP"
                                        config="Nodes/Workstation/Modules/IP/IP.xml" />
        <Module class="NetSim.Library.Modules.IcmpModule" name="ICMP" />
        <Module class="NetSim.Library.Modules.UdpModule" name="UDP" />
        <Module class="NetSim.Library.Modules.TcpModule" name="TCP" />
        <Module class="NetSim.Library.Modules.SocketModule" name="SocketModule" />
      </Node>

  <!--
     Some other nodes skipped...
  -->

    <!-- List of links used in the virtual network -->
    <Links>

      <!-- SimpleLink is used by ExternalConnection -->
      <Link name="SimpleLink" class="NetSim.Core.SimpleLink"  />

      <!-- Other Ethernet links interconnecting the nodes with the references
           to their configuration files -->
      <Link name="Eth A-D" class="NetSim.Ethernet.Ethernet100TX"
                                        config="Links/Eth A-D/Eth100BaseTX.xml"  />
      <Link name="Eth10Base2" class="NetSim.Ethernet.Ethernet10Base2"
                                       config="Links/Eth10Base2/Eth10Base2.xml"  />
      <Link name="Eth D-E" class="NetSim.Ethernet.Ethernet100TX"
                                        config="Links/Eth D-E/Eth100BaseTX.xml"  />
      <Link name="Eth10Base2a" class="NetSim.Ethernet.Ethernet10Base2"
                                      config="Links/Eth10Base2a/Eth10Base2.xml"  />
      <Link name="Eth" class="NetSim.Ethernet.Ethernet100TX"
                                         config="Links/Eth/Eth100BaseTX.xml"  />
      <Link name="Eth1" class="NetSim.Ethernet.Ethernet100TX"
                                        config="Links/Eth1/Eth100BaseTX.xml"  />
      <Link name="Eth2" class="NetSim.Ethernet.Ethernet100TX"
                                        config="Links/Eth2/Eth100BaseTX.xml"  />
      <Link name="Eth3" class="NetSim.Ethernet.Ethernet100TX"
                                        config="Links/Eth3/Eth100BaseTX.xml"  />
      <Link name="Eth4" class="NetSim.Ethernet.Ethernet100TX"
                                        config="Links/Eth4/Eth100BaseTX.xml"  />

    </Links>

  </Network>
</NetSimConfig>
```

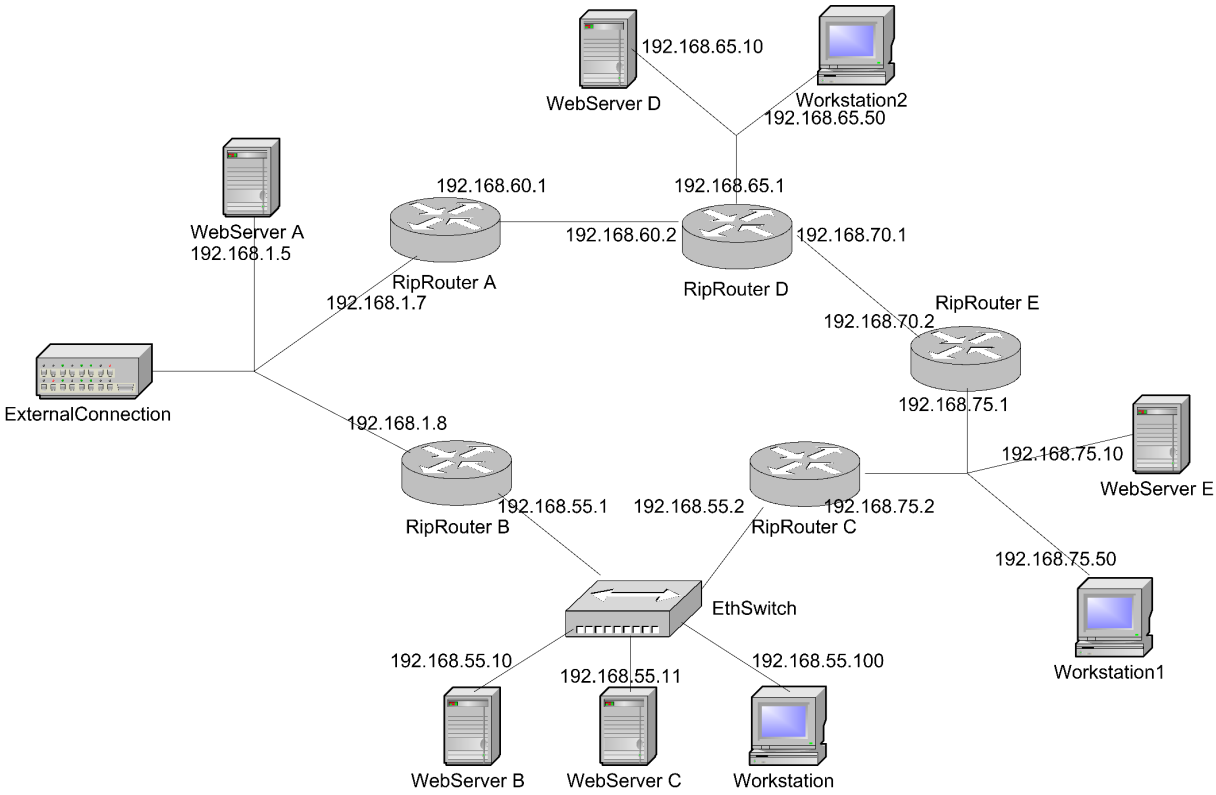


Figure 24: Diagram of an example network

Adapting the example to run on particular computer

The previous example, which is included in the simulator package, is almost ready to run on any computer. However, since it uses an interconnection to the real network, certainly, some changes are needed.

First, the name of an adapter used for capturing packets should be changed: open the configuration file for *ExtConnection* module on *ExternalConnection* node and change the contents of <Device> element to the name of the network adapter in the system. For Linux, it is usually something similar to *eth0*. For Windows, open the properties of the particular adapter and use the description on the 'General' tab. Alternatively, just run the simulator with any meaningless value, it will complain and write you a list of available adapters.

Second, change of IP addresses of three nodes that connects directly to the real network to match your subnet is required. (The current configuration is for 192.168.1.0/24 subnet.) Open the configuration files of adapters on WebServer, RipRouter A, and RipRouter B and change the IP addresses and masks to any unused values from your local subnet. Moreover, for RipRouter A and RipRouter B it is necessary to adjust those values also in configuration files of IP module and Rip application.

After all these changes are done, the simulator should be able to start the example network, reply to traceroute or ping commands, web servers respond to the http GET requests for the root page. Finally, the network should reconfigure using RIP protocol if some node or link is disabled (please wait at least 3 minutes, since it is the default timeout for entries in routing tables managed by RIP).

Appendix D – CD-ROM Contents

The following files and directories can be found on the attached CD:

`Readme.txt`
: CD-ROM general description.

`Bin`
: The directory containing compressed files that are used for installation. The installation process is described in User's Manual on page 60.

`Text/Thesis.doc`
: The text of this thesis in MS Word format.

`Text/Thesis.pdf`
: The text of this thesis in Adobe Acrobat Reader format.

`Src`
: The directory containing the source code. It can be easily opened in MS Visual Studio through `Src/NetSim/NetSim.sln`

`Doc`
: Generated documentation in HTML format.

9. Index

List of Figures