Real-Time Network Emulation with ns-2*

Daniel Mahrenholz† and Svilen Ivanov‡
University of Magdeburg
Universittsplatz 2
D-39106 Magdeburg, Germany
†mahrenho@ivs.cs.uni-magdeburg.de, ‡svilen@cs.uni-magdeburg.de

Abstract

The network simulator ns-2 implements both wireless networks and emulation *– a feature that allows to simulate a network environment among real stations. However, the real-time requirements of a network emulation introduce an inaccurate timing behavior of the simulator scheduler. These timing errors have a negative impact on the performance of network protocols in ns-2. Even more, they lead to false simulation results in the IEEE 802.11 protocol implementation. In this paper we present performance improvements in ns-2, that increase the accuracy of its virtual clock and the the exactness of the real-time simulation. Then we describe a simple time monitoring and correction technique that ensures a timely correct execution of network protocols and enables wireless network emulation in ns-2.*

1. Introduction

Real-time simulation is a modeling technique where components (simulator objects) reproduce a timing behavior similar or equal to the timing behavior of the simulated targets (simulated entities). During the development of an application it interacts with the simulated environment in the same way it would interact with a real one. This allows to test it in different environments with relatively small effort before.

The network simulator ns-2[1] is a widely accepted discrete event network simulator, actively used for wired and wireless network simulations. It has an *emulation* feature [8], i.e. the ability to introduce the simulator into a live network using a soft real-time scheduler which tries to tie the event execution within the simulator with the real-time.

However, event execution can hardly be tied to real-time without deviation. There are several factors that increase this deviation. First, if the execution of an event takes longer than the corresponding action in the real world. This often happens in the simulation of hardware devices or physical media properties. Second, by the nature of network simulation, the simulator has to execute multiple events at different nodes at the *same* time. This can hardly be achieved even with a distributed ns-2 version [12]. So they have to be serialized. And third, the system calls `write()` and `gettimeofday()` used in the current ns-2 implementation interfere with the event execution in the main loop of the scheduler because they block the simulator for a certain time period. So, normally the scheduler executes events with a delay from the real-time which may be crucial for the performance and behaviour of some protocols. The following example will illustrate this.

Consider a RTS/CTS frame-exchange sequence in the IEEE 802.11 protocol [9]. At the beginning a *transmit* event at the initiating node starts a transmission timer. When this timer expires and still a CTS frame has not been received, the node concludes that the RTS/CTS exchange has failed. However, due to delays in the event execution during this exchange, the node receives the CTS frame after the timer has expired, even if no transmission errors have occurred.

In this example (see figure 1) the transmission timer lasts $670\mu s$ and the CTS frame would arrive in the ideal case (non real-time simulation) after $666.6\mu s$. This means that a $4\mu s$ delay in event execution during this exchange would deliver the CTS frame after the timer has expired. The transmit event also schedules the start of the RTS receive at the destination node after a network propagation delay of $333ns$. However, the execution of the transmit event itself takes $98\mu s$ in this case. So, it delays the start of the RTS receive by around $140\mu s$ (this number includes the overhead of the scheduler). All the consecutive events in this exchange including the CTS receive are also delayed by this amount of time. Such a delay also occurs in the transmission of the CTS reply. Then, the transmission timer at

* This work has been supported by the German Research Foundation (DFG), grant no. NE 837/3-1

1 `http://www.isis.edu/nsnam/ns/`

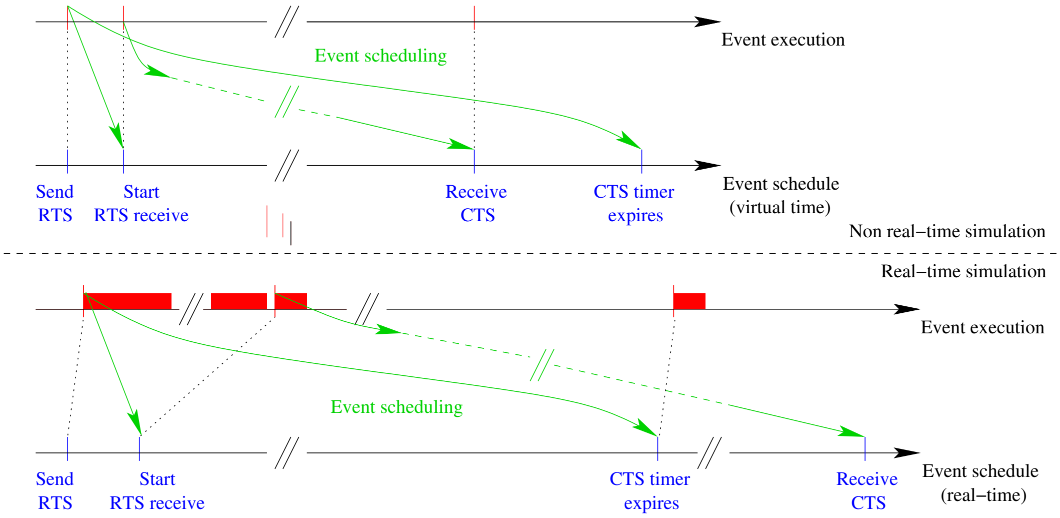

Figure 1. Sample RTS/CTS frame-exchange sequence

the initiating node always expires before the delivery of the CTS frame and the IEEE 802.11 protocol does not operate correctly. Because of the acknowledgement/retransmission scheme used in IEEE 802.11 such misleading results occur even with RTS/CTS disabled. So, delays in the event execution in a real-time emulation may change the chronological order of dependent events and falsify the execution of the simulation model.

In this paper we present changes in ns-2, that decrease the delays in the event execution and improve its accuracy of a network emulation. We first eliminate the most frequently used system calls and introduce a more precise waiting algorithm in the main loop of the scheduler. Then we present a time monitoring and correction technique for the virtual clock of the simulator. It eliminates the distribution of delays over dependent events and ensures that the simulator overhead does not change the chronological order of events. Additionally it ensures a timely end-to-end transmission of network packets if the execution of the simulation model does not constantly overloads the CPU.

The remainder of the paper is organized as follows: Chapter 2 describes our experimental setup and the properties of ns-2, on which we base our further analysis. Chapter 3 presents our performance improvements and time correction technique. Chapter 4 presents performance measurements of the original and modified ns-2. Furthermore it compares experiments with both versions to measurements in a real wireless network. Chapter 5 talks about related work and chapter 6 presents conclusion remarks and further investigations in the field.

2. Simulation Environment

In this section we describe our experimental environment – the network setup and the simulator interaction with it. Then we present key concepts of ns-2, on which we base our further investigations.

2.1. Network Setup

The *live* network in our experiments is a virtual network among a set of *User-Mode-Linux*[2] virtual machines, running on a single host. User-Mode-Linux (UML) is an extended Linux kernel, that is executed as an user-space process on a Linux host. It can be assigned physical resources separated from the host machine and connected to a network through various transport mechanisms. One of the possible transport types for UML is the TAP virtual Ethernet device[3]. It provides an interface to user-space applications through a character device (`/dev/tapXX`) and a virtual Ethernet interface (`tapXX`). In our case the user-space application is a UML virtual machine. We assign a TAP device to each UML and use it to connect the virtual host to a network. Then we create an Ethernet bridge in the host kernel, connecting all the TAP devices into an Ethernet network[4]. This structure is depicted in figure 2.

This setup enables a direct communication among the virtual machines through the bridge. But our goal is to pass this transmission through a simulated network inside ns-2, and to avoid a direct communication. So, we setup a bridge firewall[5] on the host system which blocks all packets, *for-*

2 `http://user-mode-linux.sf.net`
3 `http://vtun.sf.net`
4 `http://bridge.sf.net`

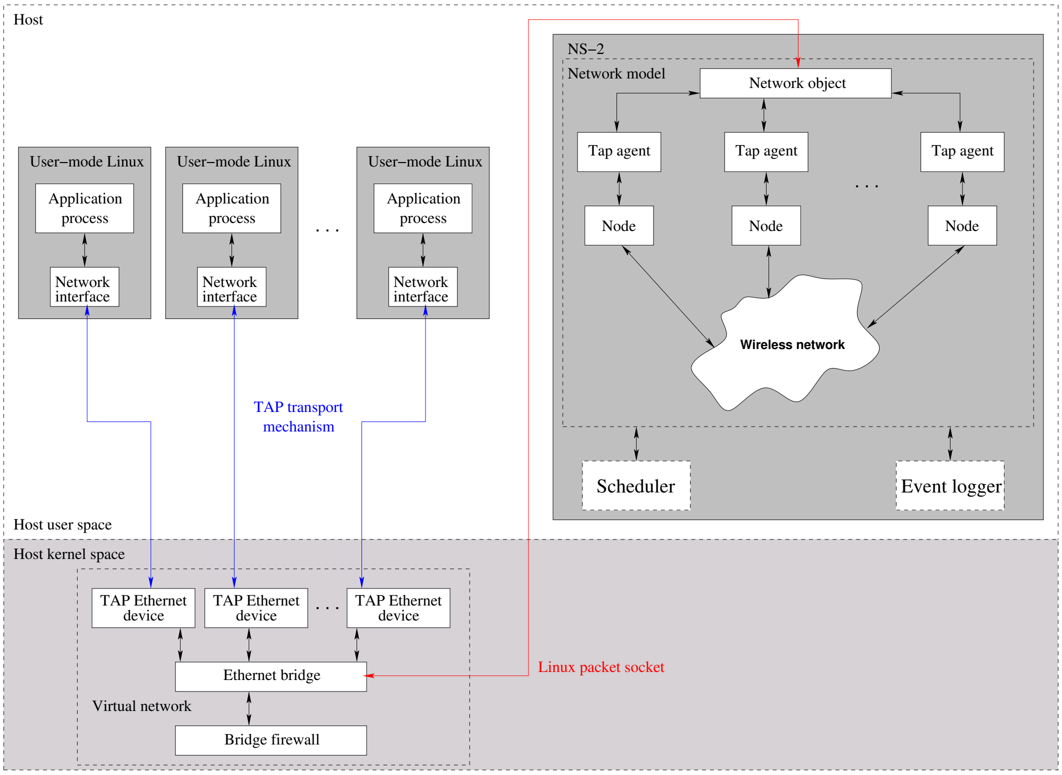

Figure 2. Experimental setup

warded through the bridge, i.e. direct transfers among the UMLs. It is now the role of the simulator to connect the virtual machines together. It acts as an emulator of a wireless network among the virtual machines and uses the bridge device as a central point for reading and writing packets from and to the virtual hosts. For this purpose, the network simulator ns-2 uses its emulation facility. It mainly consists of *network objects* and *tap agents*. The network objects are used to send and receive packets to and from a network. The tap agents are application level processes on ns-2 nodes that convert network packets between the simulated and the real network. Each tap agent can be connected to at most one network object. In our case we need a network object to access a network device on the link layer. Because ns-2 (version 2.27) does not provide such a network object in its Linux version, we implemented a new one using raw network sockets.

In our model there is a one-to-one correspondence between ns-2 nodes and virtual machines. So, we need a mechanism to map simulator nodes to virtual machines. For this purpose we use the correspondence between the Ethernet addresses of the virtual machines and the network layer addresses of the ns-2 nodes. This mapping is implemented in our new tap agent.

5 `http://ebtables.sf.net`

In the simulation model we have the following simplification: we ignore the overhead of the layers below the application layer in the virtual hosts and the overhead of the application layer in ns-2. Then we obtain a *model* of an application transmitting data over a wireless network. The overhead consists of the time spent when passing through the *"ignored"* layers and the size of the protocol headers, added to the data packets. We measured the round-trip times of ICMP echo request/reply packets over the virtual Ethernet network and over an emulated wireless network. The time spend in the virtual Ethernet was around 1.5% of the round-trip-times in the emulated wireless network. So the time overhead of the virtual Ethernet network and the protocol stack in the virtual machines is relatively small. The header overhead slightly affects the simulation because it adds to the size of a packet which is used to compute the transmission time of a packet inside the simulation. It can be avoided if we have enough information about the internal structure of the frames in the live network.

2.2. Ns-2 Schedulers

Ns-2 is a single threaded discrete event simulator. The ns-2 scheduler maintains an internal *virtual clock*. The simulator objects use this virtual clock as a time reference. The scheduler also maintains a timely-ordered list of events and processes them one by one. It takes the next earliest event from the list, advances the virtual clock till the firing time of the event, and executes it till completion. Then the control returns back to the scheduler to execute the next event. There are two basic scheduler categories that differ in the method used to advance the virtual clock – *non real-time* and *real-time*. In a non real-time scheduler, the virtual clock simply *jumps* between firing times of consecutive events. The real-time scheduler in contrast tries to execute events in the *actual* moments in real-time. It uses the physical clock of the machine as a real-time reference. If the firing moment of a next earliest event is in the future, the scheduler waits until that moment in time.

3. Ns-2 Improvements

This section describes the changes we made in ns-2 to improve its real-time behavior and enables it to run a wireless network emulation. Section 3.1 describes performance improvements, that increase the accuracy of the simulator's virtual clock. Section 3.2 presents a time monitoring and correction technique, that ensures the correct behavior of network protocols in real-time simulations with ns-2. Finally 3.3 gives an outlook on a distributed emulation setup.

3.1. Performance Improvements

Our performance improvements decrease the impact of system calls on the ns-2 process during simulation. We improve the performance of the simulator with modifications in the time measurement and waiting functions, and reduce disk I/O operations during simulation.

1. Time measurement

   The real-time scheduler in ns-2 uses the system call `gettimeofday()` to synchronize event execution with the system clock. It is the most often used system call during simulation – around $1800\, times/sec$ in a simple wireless simulation with two nodes. We replaced this system call by a function that works completely inside the user space and uses the CPU cycle counter which is available on many architectures. From this cycle counter and the CPU frequency a timestamp is simply computed as $Timestamp = \frac{\#Cycles}{Frequency}$. This approach requires platform specific instructions (**RDTSC** for Pentium compatible CPUs) but avoids the context switch delays caused by the system call. So it saves time in the main loop of the scheduler and increases the accuracy of the virtual clock.

2. Precise waiting

   Before executing the next earliest event, the scheduler has to wait for t_w. If t_w is greater that a given threshold t_t (1ms by default), the scheduler blocks itself for time t_w. Otherwise it performs busy waiting, which has a higher precision than the blocking. In the case it blocks, it wakes up with a delay d_w in time.

   The proposed waiting algorithm aims at a higher precision than the blocking method and a lower CPU consumption than the pure busy waiting approach. Instead of blocking for time t_w, it blocks for time $(t_w - d_{wmax})$ and after that does busy waiting until the event execution time. d_{wmax} is chosen such that it is greater than a considerable part of the wake up delays d_w, and it is as small as possible. A greater d_{wmax} increases the precision and CPU utilization at the same time because it causes a higher percentage of busy waiting.

3. Disk I/O reduction

   The ns-2 simulator uses disk writes to store information about occurred events. This is called *event tracing*. It is implemented as a part of the event execution. The ns-2 event tracing system maintains a memory buffer to accumulate trace information before writing it to the disk. However, the disk writes are still performed as a part of the event execution, thus blocking the simulator process for some time.

   We solve this problem by dividing the simulator into two processes. The first one is a real-time priority process which only simulates and generates event information. The second one is a low priority process that writes the event information to the trace file. These two processes communicate through a non-blocking ring buffer in a shared memory segment.

   In order to reduce disk write operations in the system during simulation, the reader process compresses the data before writing it to the output file. It uses the `zlib` compression/decompression library[6] and creates a `gzip` file in the main memory. Since the trace files contain many similar string patterns, the `zlib` library can compress them considerably (up to 10-12 times for our trace files). So, this approach significantly reduces the disk I/O during simulation or avoids it at all. The compressed storage area can be set as large as needed – but it should not exceed the available physical memory. Otherwise the operating system has to swap memory pages to disk which is an unwanted I/O operation. If the compressed trace file can not fit into the main memory I/O operations cannot be avoided. These operations increase the system load and might preempt the simulator process. This would cause longer event execution times, bigger event dispatching delays and a smaller accuracy. A possible solution to this problem is to stop the simulator process and all the processes, that are using the emulated network, for the time of flushing the buffer. Then the sys-

6 `http://www.gzip.org/zlib`

tem time could be turned back to the moment of "freezing". This approach also requires changes in the simulation scheduler to consider this time gap.

3.2. Time Correction

The performance improvements, described above, increase the accuracy of the real-time simulation in ns-2. However, the delays in the event execution cannot completely be eliminated. Furthermore, they are unpredictable under a high system load and complex simulations with a large number of nodes. So, the intended behavior of network protocols in ns-2 is still not guaranteed. In the following we describe a technique that ensures the correct execution of the simulation model, which includes the simulated protocols, even under a high system load.

Consider an event E_1 which has to be executed in a real-time instant t_{E_1}. The scheduler continuously updates its virtual clock (Vc) from the system time and waits for the moment t_{E_1}. If t_{Vc} is equal or greater than $t_{E_1}$the simulator executes the event with the delay $\Delta t_{E_1} = t_{Vc} - t_{E_1}$. Regardless how fast we query the system clock, we always obtain different values. Because of this high resolution it is unlikely to hit the exact time t_{E_1}in the waiting loop and so events are always executed with a small delay. But there is another, more critical reason for delays. The simulator can only execute one event at a time. So, if the simulator is busy with the execution of an event at t_{E_1} then E_1 has to wait until the execution of the previous event has been completed. So it will be executed with a possibly much larger delay than the one caused by the waiting loop. Independent from the reason for a delay, the results are the same and shall be explained below.

Lets assume that another event E_2 has to be executed after a time period $\Delta p_{1,2}$ after E_1 (in fact in time instant $(t_{E_1} + \Delta p_{1,2})$). The virtual clock of the scheduler is the *only* time reference for the simulator objects. They assume that all the events are executed *exactly* in the planned moments in time. Then, the event E_1 will schedule the event E_2 for $t_{E_2} = t_{Vc} + \Delta p_{1,2} = t_{E_1} + \Delta t_{E_1} + \Delta p_{1,2}$. Here the event dispatch delays interfere in the time calculations of the simulator objects. So the event E_2 in a real-time simulation is scheduled for a later time instant than in a non real-time simulation. This can lead to changes in the chronological order of events in a real-time simulation and so, falsify the behavior of network protocols.

We call an event in a real-time simulation *correctly scheduled*, if it is planned for execution at the same time instant, as if the simulation were non real-time. Let the event E_1 be the start of the simulation, planned for time 0. Then E_1 is a correctly scheduled event. However, E_2 is not a correctly scheduled event. In a non real-time simulation it would be planned for time $(t_{E_1} + \Delta p_{1,2})$ where in a real-time simulation it is planned for time $(t_{E_1} + \Delta t_{E_1} + \Delta p_{1,2})$.

Correctly scheduled events preserve the same chronological order in real-time and non real-time simulations. If all the events in a real-time simulation are correctly scheduled, then it will execute the *same* sequence of events as a non real-time simulation with the same input.

Our technique *enforces* the execution of late events in the right moments in the virtual simulation time and ensures that all events in a real-time simulation are correctly scheduled. If the scheduler observes that E_1 is late, it assigns to the virtual clock the time instant t_{E_1}, in which the event should have been executed. Then it executes the event E_1, and after its completion again updates the virtual clock from the system time. So, for the simulator objects, the events happen *exactly* in the time moments they are planned. Then, the event dispatch delays does not interfere with the time calculations and all the consecutive events that are triggered by E_1, are correctly scheduled. If we assume that E_1 is the start of the simulation, then all the events in this simulation are correctly scheduled.

This approach guarantees that a real-time and a non real-time simulations with the same input execute the same sequence of events at the same *virtual* time instants. Since the simulator objects access only the virtual clock as a time reference, the behavior of the protocols in ns-2 does not differ in both kind of simulations. Our approach has another very important effect on the actual execution of events. Compared to the virtual clock, all events are correctly executed. But compared to the system clock this is not the case. As explained before, the simulator can execute only one event at a time. So, if the execution of two events overlap, the second one is delayed until the first is finished. Now assume that E_2 schedules a third event E_3 with a difference of $\Delta p_{2,3}$. Then E_3 is scheduled relatively to the virtual clock for $t_{E_3} = t_{E_2} + \Delta p_{2,3}$. Now, if the execution of E_2 is finished before t_{E_3} than E_3 will be executed only with the delay caused by the waiting loop. If E_2 is still running at t_{E_3} but the overlap time between E_3 and E_2 is smaller than between E_2 and E_1 than E_3 is less delayed than E_2. This shows that our approach is also capable to correct errors that are caused by the sequential execution of concurrent events.

3.3. Outlook on Distributed Emulations

As explained in section 3 the increased precision also increases the CPU utilization. Additionally an increased number of virtual machines also increases the CPU utilization on the simulation host. This places strong restrictions on the scalability of the current setup. A solution that is currently under development will create a distributed emulation environment using the presented techniques. In a first step we are distributing the applications running inside the virtual machines to physical machines or virtual machines on different hosts. This highly increases the transmission time from the application to the virtual network. To handle this additional delay we will mark all packets arriving in the virtual network as already delayed so that our time correction technique can consider and compensate this. The second step then includes the distribution of the simulator itself, i.e. we will use a set of synchronized simulator instances to execute separated parts of the network. In this setup the different instances will synchronize over a separated network and we have to consider the synchronization overhead as increased event execution time that could be compensated using again the time correction technique. As we focus on clustered wireless networks we do not plan a generic solution but to place one cluster per simulator instance to minimize the synchronization overhead required for the cluster gateways.

4. Simulation Experiments

In this section we first measure the contribution of the performance improvements to the accuracy of the simulator scheduler. Then we compare the properties of an emulated wireless network and a real wireless network. All the simulation experiments, described here, use the real-time scheduler in ns-2. However, the implementation of the IEEE 802.11 protocol in ns-2.27 does not function properly in real-time simulation (as we showed in section 1). So we cannot present measurements to compare the original with the modified version.

4.1. Scheduler Accuracy

The accuracy of the real-time scheduler is the deviation Δt between the planned and the actual execution time of an event. Smaller delays mean a more accurate simulation.

Here we use a real-time simulation experiment to measure the scheduler accuracy – we compare the ns-2 versions with and without performance improvements. For this purpose we do not need network emulation, but only an experiment with the real-time scheduler. We use two wireless nodes in an ad-hoc network at 1Mbit/s data rate. One of the nodes sends packets with a constant data rate of 200Kbit/s measured on the application layer. We monitor and log the accuracy in the main loop of the real-time scheduler.

Figure 3 presents the delay distribution for both versions. We separated the delays by time and by different layers in the network stack. With only the time correction feature the application running on top of ns-2 work as expected. But the delay distribution clearly shows that we have nearly 20% of all events with a delay of $100\mu s$ or above and even some events with delays of more than 1ms. This is more than the threshold for the busy waiting and so can only happen if the

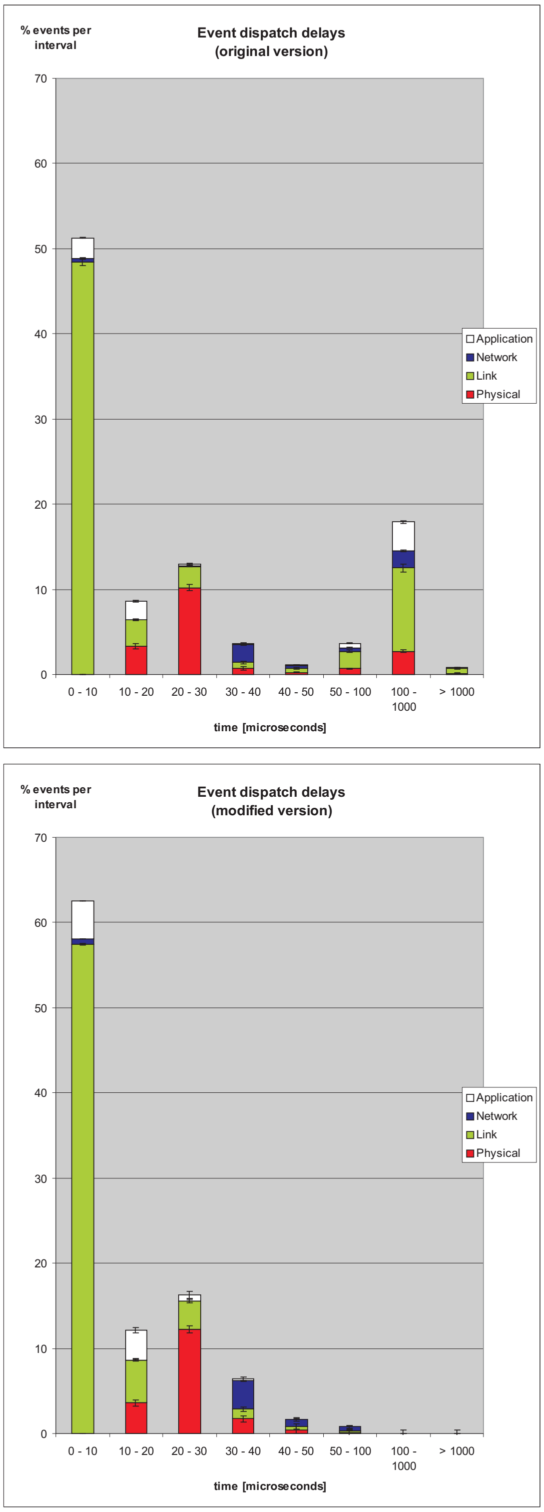

Figure 3. Scheduler accuracy – delay distribution

simulator is blocked by a system call for at least this time. We can see that about 50% of all events on the application layer are delayed by more than $100\mu s$. So we probably have an impact on the application performance in these cases. In the second diagram we see the delay distribution after applying the performance improvements. We now have an upper bound for the delay of $100\mu s$. The most important result is that we now have very small delays for all application level events which significantly reduces the probability of changes in the behavior of applications running on top of ns-2.

4.2. Network Testing

This experiment compares a real wireless network and an emulated one. This time we use the ns-2 emulation facility, the time correction technique and the performance improvements.

The network in all the measurements/experiments consisted of six wireless nodes using the IEEE 802.11b standard in ad-hoc mode at 2Mbit/s. The nodes where grouped in three pairs each with a distance of 2 meters between the nodes. Each pair used its own wireless channel to not to interfere with each other but to stress the simulator. We used the standard network testing tool – `ping` over the real and the emulated network. We run it in adaptive mode, i.e. send the next echo request immediately after receiving of an echo reply. We send at least one echo request every 10ms and counted the number of echo replies. Additionally we counted all packets with a round-trip-time (RTT) of at most 5ms as *on-time* packets.

The experiment consisted of two steps. First we measured the error rate on the physical layer (p_{Phy}) in the real network and set the same error probability in the simulator. The MAC layer in IEEE 802.11 uses an acknowledgement/retransmission scheme for delivery of frames. So, the resulting error rates at higher network levels typically do not match the physical error rate. To measure the physical error rate we send ICMP echo request/reply packets with broadcast destination MAC addresses. In this way the MAC layer does not use acknowledgements/retransmissions and we can compute the physical error rate from the results of the `ping`. Assume the ICMP error rate is $p_{ICMP} = \frac{\#Received}{\#Send}$. We only receive an echo response if both packets do not get lost. So we can compute

$$\begin{aligned} p_{ICMP} &= 1-(1-p_{Phy})(1-p_{Phy}) \\ p_{Phy} &= 1-\sqrt{1-p_{ICMP}} \end{aligned}$$

In the second step we send again ICMP echo request/reply packets, this time with unicast MAC addresses. Then we measured the achieved bandwidth and round-trip-times. The results from the experiments are shown in table 1. The numbers for the real network are the average calculated over 30 test and the root-mean-square deviation. For the emulated network we also run 30 simulation tests and calculated the values with a 95%-confidence interval. The maximum RTT spread is the difference between the minimum and maximum round-trip-times over all experiments.

	Real network	Time correction	Time cor. + perf. impr.
Bandwidth [pkt/s]	74.02 (0.02)	58.37 ($\pm$0.07)	75.43 ($\pm$0.04)
average RTT [μs]	13459.57 (2.76)	14908.24 ($\pm$23.17)	12738.53 ($\pm$4.05)
max RTT spread [μs]	77577	816030	595103
on-time [%]	99.93 (0.05)	98.65 ($\pm$0.02)	98.73 ($\pm$0.02)

Table 1. Emulated / Real network comparison

This experiment shows that the time correction method ensures the correct execution of the simulation model in a real-time simulation. We see also that the performance improvements increase the throughput of the simulator and bring the emulated network closer to the real one. Finally, the emulated network is comparable to a real network and can be used as a base for development of wireless network protocols.

5. Related Work

Network simulation is a widely used technique when developing network protocols or testing various network topologies or applications. Hence there are a lot of network simulators available. They all differ in their purposes and technologies. The majority of all network simulators are discrete event simulators that determine the network behavior on different detail levels. They work with a virtual simulation clock and so run the network model independent from the real-time. In contrast there are some simulation tools that can be integrated into live networks and therefore have to work in real-time and with real network data. To distinguish them from the event based simulators they are also called *network emulators*.

NS-2 is widely used but there are other universal network simulators like SWimNet [4], GloMoSim [3] and its commercial successor QualNet that are based on the parallel simulation environment ParSec [2]. All of them are discrete event simulators and can be used to simulate wired and wireless network setups. They provide an important method to test network protocols and topologies but share some major problems. They use their own protocol implementations an so cannot be used to test the interaction and behaviour of protocol implementations of operating systems and networks devices used in live networks. Additionally they cannot be directly used with the traffic generated by different users when running various applications. Instead they require a traffic pattern specification that is derived from a model of the user behaviour. These models use a lot of assumptions about the user and so can lead to large errors of the resulting network behaviour.

To overcome these problems in order to study actual protocol implementations under realistic conditions network emulators are required. Emulators can work in different ways. EMUNET [11] is a software library that simulates a communication network inside an user application. It combines multiple network applications that use the UNIX network interface into a single, multi-threaded application. To do so it requires minor source code extensions to the tested applications. Dummynet [13] (FreeBSD), x-SIM [5] (x-kernel), and the Hitbox pseudo-device [7] (SunOS) are kernel extensions that intercept packets on their way through the network stack to simulate the effects experienced in a real network. Dummynet works with TCP connections and simulates routers with bounded queue sizes and a given queueing policy, and communication links with a given bandwidth and delay. x-SIM and Hitbox work in a similar way and simulate end-to-end delay, packet drop and queueing characteristics. They do not require any application changes but are limited to TCP/IP traffic. A more portable solution is ONE (Ohio network emulator) [1]. It acts as a router between two network interfaces and provides adjustable transmission, queueing and propagation delays. A similar but more sophisticated solution is provided by NIST Net [6]. It is a Linux kernel module that works as a router and is optimized for high bandwidth scenarios and supports a wide range of network protocols. It emulates fixed and variable packet delay; packet reordering; random and congestion-dependent packet loss; packet duplications and bandwidth limitations. The last project that should be mentioned here is Seawind [10], a wireless network emulator. It is a generic approach to emulate various wireless networks but was only been used to test GPRS, the TCP/IP and WAP protocol, and applications using these protocols.

6. Conclusion and Further Work

In this paper we showed that the combination of a discrete event simulation with real applications in a general-purpose emulation environment is feasible even for simulated wireless networks. We introduced some changes to the widely used ns-2 simulator to optimize its behavior in the emulation mode. First a time correction feature that ensures the correct execution of the simulation model and second some performance improvements that increase the accuracy of the simulator. These changes are required to run a wireless emulation but are generic, so wired emulations will benefit from them too. In some experiments we verified that the emulation delivers similar results as a measurement in a real network.

Our future work will focus on the scalability of this emulation environment. This will include the distribution of the applications running on top of the emulation as well as the simulator itself (s. section 3.3). We also plan to introduce more changes to the internal event execution algorithm to improve the CPU utilization. This will include an optimistic ahead-of-time execution of events which already delivered promising results but requires more investigations to ensure the correct execution of the simulation model.

References

[1] M. Allman and S. Ostermann. One: The Ohio Network Emulator. Technical report, Ohio University, 1996.

[2] R. Bagrodia and R. Meyer. PARSEC: A Parallel Simulation Environment for Complex System. *Computer Magazine*, 1998.

[3] Lokesh Bajaj, Mineo Takai, Rajat Ahuja, Rajive Bagrodia, and Mario Gerla. GloMoSim: A Scalable Network Simulation Environment. Technical Report 990027, 13, 1999.

[4] Azzedine Boukerche, Sajal K. Das, and Alessandro Fabbri. SWiMNet: A Scalable Parallel Simulation Testbed for Wireless and Mobile Networks. *Wireless Networks*, 7(5):467–486, 2001.

[5] Lawrence S. Brakmo and Larry L. Peterson. Experiences with Network Simulation. In *Measurement and Modeling of Computer Systems*, pages 80–90, 1996.

[6] Mark Carson and Darrin Santay. NIST Net: a Linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):111–126, 2003.

[7] P. Danzig, Z. Liu, and L. Yan. An Evaluation of TCP Vegas by Live Emulation. In *Proceedings of ACM SIGMetrics '95*, 1995.

[8] K. Fall. Network emulation in the VINT/NS simulator. *Proceedings of the fourth IEEE Symposium on Computers and Communications*, 1999.

[9] Institute of Electrical and Electronics Engineers (IEEE), Inc. *ANSI/IEEE Std 802.11, 1999 Edition*, 1999.

[10] M. Kojo, A. Gurtov, J. Mannner, P. Sarolahti, T. Alanko, and K. Raatikainen. Seawind: a wireless network emulator, 2001.

[11] Xiannong Meng. EMUNET: Design and Implementation - A Debugging Aid for Distributed Programs in TCP/IP Based Network.

[12] George F. Riley, Richard Fujimoto, and Mostafa H. Ammar. A Generic Framework for Parallelization of Network Simulations. In *MASCOTS*, pages 128–, 1999.

[13] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.