Adjusting the ns-2 Emulation Mode to a Live Network*

Daniel Mahrenholz[1] and Svilen Ivanov[1]

University of Magdeburg, Universitätsplatz 2, D-39106 Magdeburg, Germany,
mahrenho

Abstract. The network simulator ns-2 implements both wireless networks and *emulation* – a feature that allows to simulate a network environment among real stations. In contrast to a discrete event simulation an emulation is affected by various system parameters. The understanding of the impact of those parameters is crucial to obtain valid results from an emulation experiment. In this paper we evaluate a modified version of the ns-2 that has been optimized for emulations. We analyze the effect of different parameter settings in various experiments and compare the results with measurements in a live network.

1 Introduction

Real-time simulation (*emulation*) is a modeling technique where simulator objects reproduce a timing behavior similar or equal to the timing behavior of the simulated entities. This technique is used to simulate a desired application environment during an application development process. So it allows testing of different simulated environments with relatively small effort. The main goal is that an application can interact with the simulated environment during the development phase the same way as it would with the real one and so can be deployed later without modification.

The network simulator ns-2 [12] is a widely accepted discrete event network simulator, actively used for wired and wireless network simulations. It has an *emulation* feature [7], i.e. the ability to simulate an arbitrary network in real-time. Discrete event simulation assumes that event execution does not take any (simulation) time. In [10] we showed the problems that arise when the simulation clock runs in real-time. This is because event execution times now have to be considered. We also explained the weaknesses of the current ns-2 implementation that lead to an unpredictable timely behavior and so result in wrong simulation results (especially for wireless networks). We presented some generic modifications to the ns-2 to solve these problems and to highly improve its emulation abilities (s. section 2.1 for a brief overview). In this paper we will study the effects of various system and simulation parameters on the performance and

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

correctness of the simulation results. We also take a closer look onto the scaling limits of our approach.

The remainder of our paper is structured as follows. Section 2 describes our emulation environment – the hard- and software setup we are using and the modifications made to improve the ns-2 emulation mode. Section 3 investigates various parameters of the emulation environments and their impact on the performance of the simulator and the accuracy of the emulation results. Section 4 discusses related projects and finally section 5 gives some conclusions and an outlook on further work.

2 Emulation Environment

Our emulation environment consists of a host running a modified version of the ns-2 simulator (s. section 2.1) and a number of Linux machines, one for each node in the simulation. The *live* network in our setup is a virtual network among the Linux machines running the node software. To reduce the number of physical Linux hosts needed for an experiment we use *User-Mode-Linux* (UML) virtual machines. UML is an extended Linux kernel which runs as an application process on top of a Linux host. It has its own network interface and appears as a full-featured operating system for an application running inside. Figure 1 depicts the current setup.

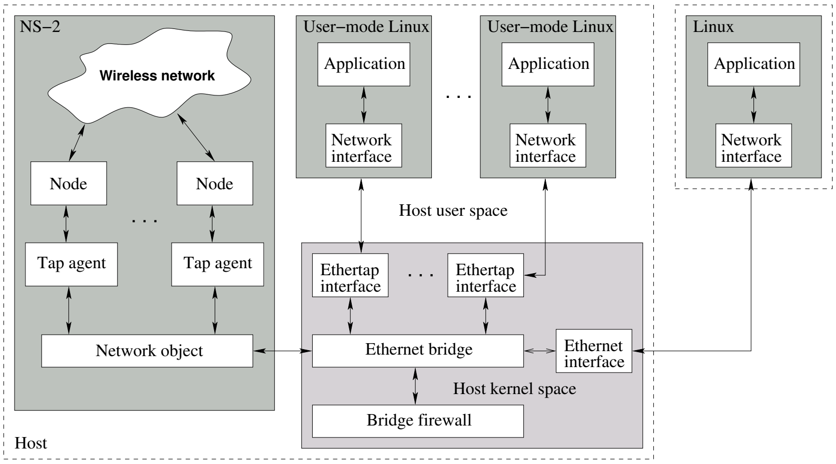

Fig. 1. Emulation Environment Setup

To connect the Linux machines we use separated Ethernet interfaces for the physical hosts and the TAP virtual Ethernet device for the UML machines. To build the virtual network we created an Ethernet bridge in the ns-2 host kernel that connects all TAP devices and the physical Ethernet device into a single Ethernet network. This virtual network enables all machines to communicate directly with each other. But our goal is to route all traffic through the simulation. So we have to block direct communication using a firewall and only allow traffic through the simulation.

2.1 NS-2 Modifications

Ns-2 already provides an emulation facility, but it proved to be unsuited for the emulation of wireless networks because of various reasons. The major reason is, that event execution times are not considered in the real-time scheduler used for the emulation mode. Additionally the simulator makes excessive use of operating system functions inside the real-time critical event execution loop for the time synchronization and the event logging. These system calls can block the simulator for an unpredictable time – often longer than the timeouts defined in different protocols. This falsifies the execution of the simulation model or even makes it impossible – especially for wireless networks.

Implementation Improvements: This section details the changes made to the ns-2 to improve its overall performance and to increase the accuracy of the virtual simulation clock. These changes can be divided into performance improvements and some changes to the execution of the simulation model. A more detailed explanation can be found in [10].

Time measurement: To synchronize event execution with the system time the ns-2 real-time scheduler uses the system call gettimeofday(). This system call is called very often during a simulation which consumes a lot of time. So we replaced this system call by a function that uses the CPU cycle counter to measure times. Together with the CPU frequency we get $Timestamp = \frac{\#Cycles}{Frequency}$. Such a cycle counter is available in many current CPUs. Of course our approach is only feasible for CPUs that run on a constant frequency or can be configured to do so which is possible in most cases.

Precise Waiting: Before executing the next event the real-time scheduler has to wait until the moment in time the event was planned for. For this it uses a combination of busy waiting and blocking. If the time to the next event t_w is below a certain threshold t_t (1ms by default) the scheduler performs busy waiting. Otherwise it sleeps for t_w. The problem now is that the process does not regain control of the CPU exactly at the planned time but with a delay of d_w. To increase the precision we now change the sleeping time to $(t_w - d_{wmax})$. We chose d_{wmax} such that it is greater than a considerable part of the wakeup delays and perform busy waiting afterwards. This additional busy waiting introduces a higher CPU utilization and so there is a tradeoff between precision and CPU load (s. section 3.4).

Disk I/O reduction: The simulator stores information about events happening during a simulation to disk. This event logging is implemented as part of the event execution. Although the simulator maintains a memory buffer to accumulate trace information before writing them to disk, disk writes are still performed as part of the event execution. These I/O operations block the simulator process for a certain time and so can delay the execution of events. To solve this problem we divided the simulator into two processes – a high priority process that executes the events and a low priority process that collects the trace data and stores them to disk. Both communicate through a non-blocking ring buffer in a shared memory segment. So the logging cannot longer block the simulator directly. But the disk I/O still can interfere with the execution of the simulator process. So in order to reduce the number of write operations during the simulation or to avoid them at all in many scenarios, the logging process compresses the data before storing it using the zlib library.

Time Correction: The performance improvements described in the previous section increase the accuracy of real-time simulations and increase the overall event throughput. But these improvements cannot avoid delays in the event execution, especially under a high system load or when simulating complex scenarios. Furthermore, regardless how big these delays are, they all lead to an incorrect execution of the simulation model. This and our proposed solution will be explained in the following.

The simulation model is executed using the time of the virtual clock (t_{Vc}) which is constantly updated in the main loop of the simulation scheduler. Everytime the scheduler checks whether it can execute a new event from the event queue, it reads the current time from the system clock and updates the virtual clock. Now consider an event E_1 that should be executed at t_{E_1}. The scheduler waits until that moment and then executes the event. Because of the high resolution of the used timer, the event is executed with a measurable delay at $(t_{E_1} + \Delta t_{E_1})$. Now assume that E_1 schedules another event E_2 with a time difference of $\Delta p_{1,2}$. Because all simulation objects assume that all events are executed at the exact moments in time, events are scheduled relatively to the current time (t_{Vc}). But compared to the system clock E_2 is scheduled for $(t_{E_1} + \Delta t_{E_1} + \Delta p_{1,2})$ and will be executed at $(t_{E_1} + \Delta t_{E_1} + \Delta p_{1,2} + \Delta t_{E_2})$. This clearly shows that the current method accumulates the delays inside a chain of dependent events. This delay accumulation is even more serious in overload situations. If we have more than one concurrent chain of events then it is likely that the delay in different chains increases differently. This can lead to changes in the chronological order of events and so falsifies the simulation model. To solve this problem we use the planned execution time that is stored inside the event object. When an event is executed, we set t_{Vc} to the planned time. So dependent events will be scheduled relatively to the planned execution time defined by the simulation model. This does not avoid a late execution, compared to the system clock, but prevents the accumulation of errors. Furthermore, if the CPU is not constantly overloaded, it reduces previous delays.

3 Emulation Tuning

Before we can tune the parameters of the emulation environment we have to understand their impact on the setup. In this section we will first explain the experimental setup used to run all the tests. Then we will look on the parameters that need to be determined and setup in order to adjust the emulation to the behaviour of a live network. Afterwards we discuss the impact of various parameters on the performance of the simulator and the accuracy of the emulation results. From this we finally derive some guidelines to handle the tradeoff between accuracy and performance.

3.1 Experimental Setup

Figure 1 already showed the principle experimental setup. Now we will have a closer look onto the used hard- and software. All emulation experiments were run using the User-Mode-Linux machines. On the one hand this requires the most resources on the simulation host but on the other hands allows the most accurate measurements. This is because of two reasons. First we have a highly precise and perfectly synchronized clock in all components because all access the cycle counter of the same CPU. Second we can ignore the transmission overhead caused by the virtual Ethernet. We measured the overhead and it was below 1.5% of the end-to-end transmission time. Our simulation host was a AMD XP 2400+ with 512 MB of RAM running the vanilla 2.6.1 Linux kernel. We used ns-2 version 2.27 as the basis for our modifications. We setup all the UML machines to use at most 32MB of RAM and the logging process of the simulator to use 100MB for the compressed trace file. So we avoided all disk I/O operations during a simulation run because all trace data fit into the compressed in-memory file. Inside the UML machines we used the same 2.6.1 Linux kernel as on the host system.

For the live-network measurements we used different combinations of laptops running Linux kernel version 2.4.25 and 2.6.3 with different 802.11b wireless network cards to compensate for driver specific effects. Our traffic generator was a slightly modified ping version. We added a command line option to specify a deadline for the receiving of the response packet and some statistic extensions. All packets that meet the deadline are counted as *on-time packets*.

3.2 Initial Parameter Setup

The first step to adjust the emulation to the real-world is a careful parameter setup. The most important parameters are the transmission bit rate, the error probability, the transmission power, and the retransmission counters. Additionally the node distance and the traffic pattern are important for the simulation scenario. The transmission bit rate, transmission power, and retransmission counters are values that can be simply specified in the simulator setup and on the network interface. For all experiments we used a bit rate of 2Mbit/s and set the transmission power and retransmission counters to the standard values reported by the wireless network adapter. The node distance was set to 2m in our tests. The value for the error probability can not be setup that easy. It has to be measured in the real world and then can be specified in the emulation.

Ns-2 allows to generate errors on the sender or the receiver side. Because we try to model errors on the wireless medium we decided to generate errors at the receiver. So a packet send as broadcast is not received by all or none nodes but by any possible number. To measure the error rate we use ICMP echo request/response packets and count transmitted and received packets. To measure the raw packet loss rate we have to make sure that there are no retransmissions on the MAC layer. An effective way to enforce this is to use broadcast packets. The ping command can send echo requests to a broadcast address, but normally responses are send to the unicast address of the requesting host. To circumvent this behaviour we specify a broadcast MAC address on both hosts as the MAC address of the opposite host. So when using a ping with an unicast IP address, the packets are send as broadcasts in both directions. With this setup we send 100.000 packets with a rate of 100 per second and counted the received packets. From these numbers the raw packet error rate can be computed. Assume the ICMP error rate is $p_{ICMP} = \frac{\#Received}{\#Send}$. We only receive an echo response if both packets does not get lost. So we can compute

$$p_{ICMP} = 1 - (1 - p_{raw})(1 - p_{raw})$$
$$p_{raw} = 1 - \sqrt{1 - p_{ICMP}}$$

We ran several measurements with small (8 byte payload) and large (1000 byte payload) packets in an undisturbed and a normal office environment and obtained the numbers shown in table 1 that then were used for the following experiments.

Environment	p_{ICMP}	p_{raw}
office, small packets	0.046%	0.022%
office, large packets	0.6%	0.3%
undisturbed, small packets	0%	0%
undisturbed, large packets	0%	0%

Table 1. Wireless error probabilities

As the first experiment we ran a ping with an adaptive transmission rate of at least 100 packets per second. Adaptive means that we send a new packet at least every 10ms or immediately after receiving a response before this time. This way we performed a bandwidth measurement. Because of the adaptive sending the value can be above 100 packets per second. We additionally specified a deadline of 5ms for this test and counted all packets as *on-time packets* where we got a response within this time. The deadline has no effect on the trasmission of packets, it is only used for statistics. Table 2 shows the result for the measurement with small packets and no RTS/CTS (s. section 3.3 for more details). Measurements for large packets and other parameter settings will be presented below.

All emulation test were run with the method of independent replications. The presented numbers for the live network are the average calculated over 30

	Emulation	Live network
Bandwidth [pkt/s]	286.85 ($\pm$0.290)	307.17 (14.58)
average RTT [ms]	3.34 ($\pm$0.003)	2.77 (0.017)
on-time [%]	99.84 ($\pm$0.000)	99.66 (0.180)

Table 2. Comparison Emulation/Live network ping

tests and the root-mean-square deviation (its square-root for better readability in brackets). For the emulation we give a 95%-confidence interval also calculated over the 30 different tests. The bandwidth is the number of successfully transmitted packets per second. The average round-trip-time (RTT) is the average time difference between sending the request packet and receiving the response. Each test run for either at least 10 minutes or 100.000 packets to get a broad result set.

The results show that we get comparable results in the emulation and in the live network. The throughput is slightly smaller in the emulation and we have larger runaway results. We can also see that the confidence interval in the emulation is small.

3.3 Parameter Impact on the Simulator Performance

In this section we will take a closer look on the effects that different parameters have on the performance and the results of the emulation. We have chosen the scenarios so that we could verify all the results in a real-world measurement. Understanding these effects is crucial to get usable results from an emulation because choosing a wrong set of parameters can falsify an emulation. So the following experiments should help to identify an erroneous simulator behaviour. The next chapter will then discuss possible corrections to obtain usable results at the cost of lesser accuracy.

The first parameter we want to discuss is the use of the RTS/CTS frame exchange sequence in the 802.11b MAC layer. The RTS/CTS mechanism tries to overcome the hidden-station problem. If two senders are far away, but want to sent to the same or colocated stations, they cannot sense each other on the medium due to being out of range of each other, so their transmissions would collide at the receiver. With RTS/CTS, the sender issues a small RTS frame, which is answered by the receiver with a small CTS frame. As both frames contain the length of the transmission that is about to be started, all receivers of any of these frames know about this transmission and do not interfere with it.

To measure the effect of the RTS/CTS usage we used the modified ping with large packets (1000 byte payload) and an adaptive transmission rate with at least 100 packets per second. Because of the large packet size we increased the deadline to 15ms. The raw error rate was set to the one measured in the office environment (0.3%). We ran both measurements with and without RTS/CTS each 30 times. We used large packets for this experiment because RTS/CTS makes no sense for small packets in a live network.

The table 3 shows the results from this experiment. In both cases the emulation delivers a slightly higher throughput than the live network. When we

	CTS/RTS enabled		CTS/RTS disabled	
	Emulation	Live network	Emulation	Live network
Bandwidth [pkt/s]	78.17 (±0.030)	74.02 (0.020)	86.24 (±0.020)	80.36 (1.130)
average RTT [ms]	12.55 (±0.002)	13.46 (0.003)	11.42 (±0.001)	12.38 (0.020)
on-time [%]	99.83 (±0.010)	99.93 (0.050)	99.86 (±0.000)	99.99 (0.020)
CPU utilization	44.82 (±0.020)	—	39.67 (±0.020)	—

Table 3. 1000 byte payload; 15ms deadline; office environment

look back to the results in table 2 we see that the emulation delivers a lower throughput for very small packets. So for the average case the throughput will be the same in both scenarios. The on-time packets are almost the same in all measurements. Furthermore we can see that the use of RTS/CTS has the same effect in the emulation and the live network. In both cases it results in an about 10% decrease of the throughput. But we can also see that the use of RTS/CTS increases the CPU utilization of the simulator by about 12% due to the fact that it now has to simulate a larger number of events.

The second experiment tests the effect of different error rates. We compare the measurements of the previously used office scenario (0.3% error rate) with an artificial scenario (*Noisy)* with a much higher value of 5%. For both scenarios we run the adaptive ping test with small and large packets and a 15ms deadline.

	8 byte payload		1000 byte payload	
	Office (0.3%)	Noisy (5%)	Office (0.3%)	Noisy (5%)
Bandwidth [pkt/s]	286.85 (±0.290)	240.27 (±0.130)	86.24 (±0.020)	74.08 (±0.060)
average RTT [ms]	3.34 (±0.003)	3.69 (±0.001)	11.42 (±0.001)	12.60 (±0.008)
on-time [%]	99.84 (±0.000)	99.78 (±0.000)	99.86 (±0.000)	81.92 (±0.100)
CPU utilization	80.2 (±0.070)	74.75 (±0.090)	39.67 (±0.020)	37.45 (±0.020)

Table 4. no RTS/CTS; 15ms deadline

The table 4 shows the results for these measurements. We notice several things in this experiment. First that the throughput decreases – an expected result because we lose more packets. Second that the average RTT increases. This is due to the fact that we now have more retransmissions on the MAC layer. As a consequence the percentage of on-time packets drops in the second measurement because the average is already short below the deadline. A more surprising result is the reduced CPU utilization. It seams that the reduced overall number of transmitted packets outweighs the simulated retransmissions which increase the number of events to simulate.

The third test will investigate the effects caused by one of our modifications to the ns-2 simulator. As described in section 2.1 we introduced a correction time to compensate for imprecise system calls – the blocking sleep in our case. We reduce the blocking time and use busy waiting up to the starting time of an event. This busy waiting increases the CPU load. As a result other processes in the system receive less CPU time to work because the simulator process runs with a high priority. This especially effects the UML machines running on the same host because they are blocked by the simulator. We ran the test with large packets, no RTS/CTS and a deadline of 15ms. To get a high precision we set

d_{wmax} (s. section 2.1) to 2ms. Then we reduced this value to 0.2ms which results in a lower precision.

	high precision	low precision
Bandwidth [pkt/s]	86.24 (±0.020)	88.02 (±0.030)
average RTT [ms]	11.42 (±0.001)	11.18 (±0.002)
on-time [%]	99.86 (±0.000)	99.86 (±0.010)
CPU utilization	39.67 (±0.020)	5.25 (±0.020)

Table 5. no RTS/CTS; 1000 byte payload; 15ms deadline

The results from this experiment are shown in table 5. From this values we clearly see that the low precision settings result in a significantly lower CPU utilization. All other values are almost equal. So the lower precision of the emulation does not effect our ping application. In section 3.4 we take a closer look on the internal behaviour of the simulator and show that this setting really results in a lower precision although it can not directly be seen in these results.

The last experiment will test the scalability of our solution. Now we increase the number of nodes from two to six and run the same ping tests pairwise. In the simulation we configured the three node pairs to use different wireless channels. So they should not interfere with each other and the results of each pair should be comparable to the two node scenario. As in the third test we run the six node scenario twice, first with a high precision and second with the lower one. In both cases we transmitted large packets using RTS/CTS and compare the results with the two node measurement in the live network (s. table 3).

	high precision		low precision	
	6 nodes	2 nodes	6 nodes	2 nodes
Bandwidth [pkt/s]	45.41 (±0.070)	74.02 (0.020)	75.43 (±0.040)	74.02 (0.020)
average RTT [ms]	18.54 (±0.014)	13.46 (0.003)	12.74 (±0.004)	13.46 (0.003)
on-time [%]	9.64 (±0.010)	99.93 (0.050)	98.73 (±0.020)	99.93 (0.050)
CPU utilization	73.94 (±0.060)	—	45.12 (±0.060)	—

Table 6. RTS/CTS; 1000 byte payload; 15ms deadline

The results of this experiment can be seen in table 6. The first measurement delivers results completely different from the live network. The bandwidth is roughly the half and the on-time packets are only a tenth. The CPU utilization is also very high. This clearly shows that we caused an overload situation. The second measurement with a lower precision in contrast delivers results that match the reference values. The reasons for these results will be discussed in the next section. One might wonder why six nodes do not work at a CPU utilization of the simulator of about 75% because table 4 clearly shows that we run a valid emulation with 80%. The answer is simple if we look on the remaining CPU time available for the UML machines. In the previous experiment the remaining 20% were distributed to two machines so each virtual machine could use about 10%. In this experiment the remaining 25% distributes to six virtual machines – about 4% for each, which obviously is not enough. With the decreased precision we now have about 9% per virtual machine left which is sufficient.

3.4 Trading Accuracy for Performance

As seen in the previous section we can improve the simulator throughput and so the scalability by adjusting the compensation (safe) time for imprecise system calls. We also said that this reduces the precision of the emulation. By only looking at the results of the experiment this statement is questionable. So we will present some more measurements that will prove it. With the precision or accuracy of the emulation we mean the difference between a planned and the actual execution of an event. The planned time is the time defined by the simulation model. To measure the difference between both times we extended the scheduler to record the planned execution time and the time of the system clock of every event that it executes. We ran this measurement for the two and six node scenario. The results look similar but we will only discuss it for the six node scenario because it makes an important difference there.

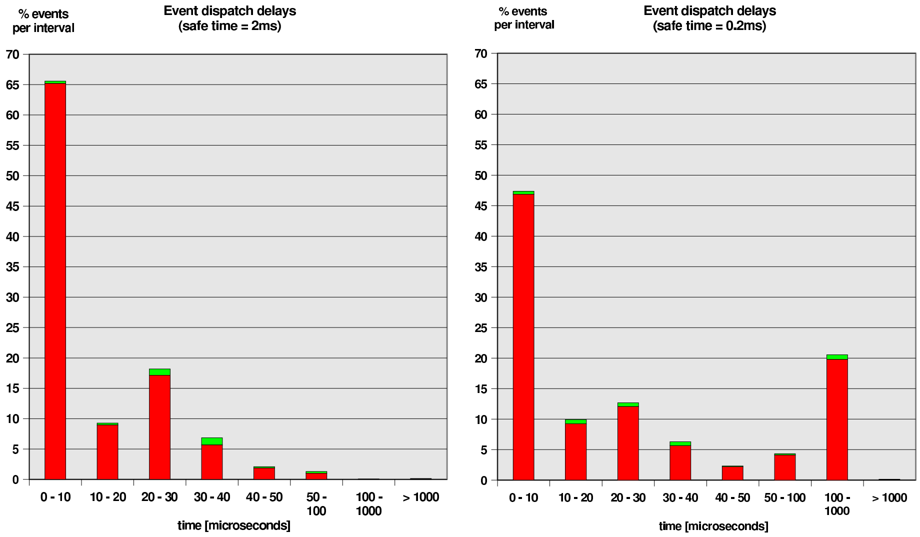

Fig. 2. Delay distribution with high/low safe time

The figure 2 shows the distribution of delays. With the high safe time we have an upper bound of $100\mu s$ and more than 90% of all events with a delay below $30\mu s$. With the reduced safe time we still have about 67% of all events delayed less than $30\mu s$ but now 28% with a delay of $100\mu s$ up to 1ms and even some events with a delay of more than 1ms. For our applications such delays are not a serious problem. But transmitting packets with larger delays to the receiving application increases the likelihood of changes in the application behavior. So the application running on top of the emulation system defines bounds for the precision of the simulator.

After looking at the results of this measurement we will explain the reasons for this simulator behaviour. A high CPU utilization is mainly caused by the busy waiting inside the scheduler. So reducing the busy waiting time decreases the CPU utilization. Busy waiting is required if the execution of the next event is scheduled for a time below the threshold where we safely could use a blocking wait and some busy waiting afterwards. Increasing the number of nodes not only increases the number of packets to simulate. It also significantly increases the probability that the starting time of two consecutive events differ less than the safe value. So just placing a new event between two others can convert a lot of sleeping time into busy waiting time which highly increases the CPU utilization. As a side effect this leads to less remaining CPU time for the virtual machines because they are blocked during the busy waiting of the simulator. So the busy waiting places restrictions on the event throughput and on the scalability of our approach especially when using multiple virtual machines on the same host. Solutions to this scalability problem are the use of multiple CPU systems and the distribution of the applications to external hosts.

4 Related Work

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. NS-2 is widely used but there are other universal network simulators like SWiMNet [2], GloMoSim [6]. They provide an important method to test network protocols and topologies but share some major problems. They use their own protocol implementations and so cannot be used to test the interaction and behaviour of actual protocol implementations used in operating systems and network devices used in live networks. Additionally they cannot be directly used with the traffic generated by different applications. To overcome these problems in order to study actual protocol implementations under realistic conditions network emulators are required [8]. Emulators can work in different ways. EMUNET [11] is a software library that simulates a communication network inside an user application. Dummynet [13] (FreeBSD), x-SIM [3] (x-kernel), and the Hitbox pseudo-device [5] (SunOS) are kernel extensions that intercept packets on their way through the network stack to simulate the effects experienced in a real network. 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 [4]. 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 [9], 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.

5 Conclusion and further work

In this paper we showed that the combination of the flexibility of a discrete event simulation with real applications in a general-purpose environment is feasible even for simulated wireless networks. We presented some changes to the widely used ns-2 simulator to optimize its behavior in the emulation mode. In several experiments we measured the impact of different parameter settings on the performance of the emulation system and the accuracy of the measured results. We proposed a method to configure the simulation using real-world parameters and verified by experiment that the emulation behaves the same way as the live network. We also recognized that our current setup imposes some tight limitations on the scalability of the simulation complexity. So future work will mainly focus on a distributed setup. In a first step we distributed the applications running on top of the simulated network to separated hosts. This already significantly increased the number of possible nodes in the simulation. Later we will study the distribution of the simulator itself, either by a separation and distribution of independent parts of a simulated network to different simulator instances (as in the six node scenario) or possibly by applying our modifications to the parallel distributed version of ns-2.

References

1. M. Allman and S. Ostermann. One: The Ohio Network Emulator. Technical report, Ohio University, 1996.
2. 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.
3. Lawrence S. Brakmo and Larry L. Peterson. Experiences with Network Simulation. In *Measurement and Modeling of Computer Systems*, pages 80–90, 1996.
4. Mark Carson and Darrin Santay. NIST Net: a Linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):111–126, 2003.
5. P. Danzig, Z. Liu, and L. Yan. An Evaluation of TCP Vegas by Live Emulation. In *Proceedings of ACM SIGMetrics '95*, 1995.
6. Lokesh Bajaj et al. GloMoSim: A Scalable Network Simulation Environment. Technical Report 990027, 13, 1999.
7. K. Fall. Network emulation in the VINT/NS simulator. *Proceedings of the fourth IEEE Symposium on Computers and Communications*, 1999.
8. S. Floyd and V. Paxson. Difficulties in simulating the Internet. *IEEE/ACM Transactions on Networking*, 9(4):392–403, August 2001.
9. M. Kojo, A. Gurtov, J. Mannner, P. Sarolahti, T. Alanko, and K. Raatikainen. Seawind: a wireless network emulator, 2001.
10. Daniel Mahrenholz and Svilen Ivanov. Real-Time Network Emulation with ns-2. In *Proceedings of DS-RT'04*, Budapest, Hungary, October 2004.
11. Xiannong Meng. EMUNET: Design and Implementation - A Debugging Aid for Distributed Programs in TCP/IP Based Network.
12. The Network Simulator ns-2. http://www.isi.edu/nsnam/ns/.
13. Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.