Technical Issues of Real-Time Network Simulation in Linux

Andrei V. Gurtov

Department of Computer Science, University of Helsinki
Department of Computer Science, University of Petrozavodsk

P.O.Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland

E-mail: gurtov@cs.helsinki.fi

Abstract

Real-time network simulation requires dealing with miscellaneous technical problems to achieve a correct and timely execution. Ignoring those issues can render a valid model useless, because its implementation would produce erroneous results. This paper identifies and discusses the problems specific for a Linux operating system on the x86 architecture. A problem of accurate event scheduling in a simulation process without disturbing other processes is the most important and is considered in detail. Several solutions to this problem are evaluated by measurements. The results show that no single solution fits all criteria, but the most appropriate method can be selected according to the goals of a simulation study.

If you're trying to solve real-time sort of problems,
you are dealing with some fairly thorny technical issues.

B. Gallmeister, a vice-char of POSIX.4. [6]

1 Introduction

Studying the behavior of Internet protocols over a real data link or network is often costly or, if a system is only in a development stage, impossible. An alternative way is to build a model that emulates the network of interest and then using this model to measure the performance of real networking applications.

An understandable desire of any modeler is to concentrate the effort on developing a conceptual model of the system under study and to treat the computer as a perfect implementation tool that accurately follows the event schedule. Unfortunately, this does not work, as most off-the-shelf personal computers and UNIX-like operating systems are not designed for real-time use, have coarse timer resolution, and are prone to delays caused by the hardware (a disk or network access) and by the operating system. Especially in a multi-process environment, keeping a real-time schedule can be hard, because a simulation process has to compete with other processes for system resources.

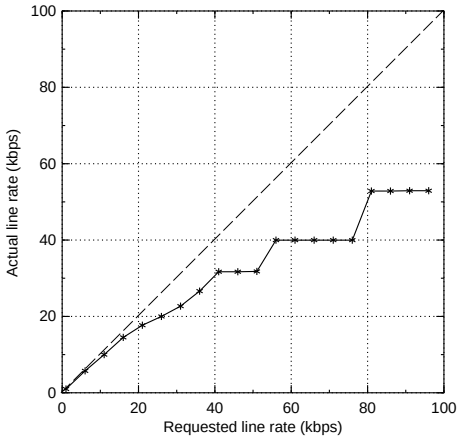


Figure 1. *Actual versus requested line rate. Measured with WINES simulator using 100-byte packets. Sleeps are performed using a standard Linux system call*

Consider Figure 1, for example. It presents performance results from the first version of the Wireless Network Simulator (Wines), a tool for studying the behavior of network protocols over GSM, developed at the Department of Computer Science, University of Helsinki. Wines emulates

a slow wireless link by delaying data packets, and the actual line rate maintained by the simulator is expected to be the same as requested in a configuration file. In practice, as can be seen from the figure, the actual line rate is lower than the requested line rate. The error is produced because the simulator relies on a standard Linux system call to perform accurate delays.

Appropriate services of an operating system for real-time applications is an active research area. An important landmark is POSIX.4 specifications for portable real-time programming [6]. However, many related issues are highly specific for a particular hardware and operating system.

In this paper we discuss technical issues of real-time network simulation on a Linux operating system run on a PC[1]. A problem of accurate event scheduling in a simulation process without disturbing other processes is the most important and is considered in detail. Most related work is concentrated only on achieving the highest possible accuracy, but ignoring practical factors that are sometimes decisive for the usage of a method. In this paper, we take into consideration such issues as the amount of modifications needed in the Linux kernel, and transparency of a method for applications.

Several solutions to the problem of accurate delay are evaluated by measurements. The results show that no single solution fits all criteria, but the most appropriate method can be selected according to the goals of a simulation study. Other problems are outlined and possible solutions to them are suggested, but an extensive evaluation is the subject of future work. Details not present in this paper due to the lack of space can be found in [8].

2 Seawind real-time simulator

A Software Emulator for Analyzing Wireless Network Data transfers (Seawind) is developed as a tool for exploring the behavior of real Internet protocols (mostly TCP) over wireless datalink services provided by GSM, GPRS, and HSCSD [9]. It may be classified as a real-time distributed functional simulator [2]. The simulation system consists of several simu-

[1]We use the term PC to refer to any personal computer based on i386 and its successors

lation processes connected in a pipeline, so that every simulation process corresponds to some subsystem of the modeled network. Simulation processes can be distributed on several computers and exchange messages using unmodified TCP or UDP protocols.

The simulation process is designed based on the Mowser library [1], that among other tools includes a generic event dispatcher (**mev**). A Mowser client can register event handlers for a number of specific events (a descriptor is ready for writing or reading, an alarm goes off, a process receives a signal, etc.). Unfortunately, mev was not initially designed to be a real-time scheduler and was never used in this way. Experience with Seawind will show the existing problems, and appropriate enhancements could be made to mev in the future.

Several simulation processes are managed with a control tool via a graphical user interface. The client and the server are normal Internet hosts that run a networking application over the Seawind system that tunnels packets possibly delaying, modifying or dropping them. The background load can be emulated either artificially or explicitly with external load generators. The configuration of the simulation process is read before starting a test and is not a problem, but logging may happen during an experiment run and can cause undesired delays.

Two factors imply that it is not wise to demand the usage of a modified Linux kernel for all experiments. First, as a rule, every simulation process should be run on a separate PC. Second, the Seawind simulator is used in several organizations and they may not have resources to install a modified Linux system. In this paper we outline the cases in which the kernel modification is a must, and cases where the required accuracy can be achieved by suggested methods in the user software.

3 The problem of an accurate sleep time

3.1 Definition of the problem

The standard Linux kernel on PC provides a process sleep time resolution of 10 ms with a minimum of approximately 20 ms. As a rule, the actual sleep time is 10 ms more than requested. In the later sections we will see reasons for such coarse behavior, but first we consider the implications of these facts to our real-time network simulator.

Figure 2 shows the delay per packet to emulate a slow link of a given line rate. The delay value is determined by the line rate and by the packet size. To demonstrate limitations of the standard Linux sleep method, let us consider modeling a GPRS data link. Conceptually, three main levels of model granularity can be identified: the IP packet layer (typical packet size of 1000 bytes), LLC (typical packet size of 200 bytes), and RLC (typical packet size of 25 bytes).

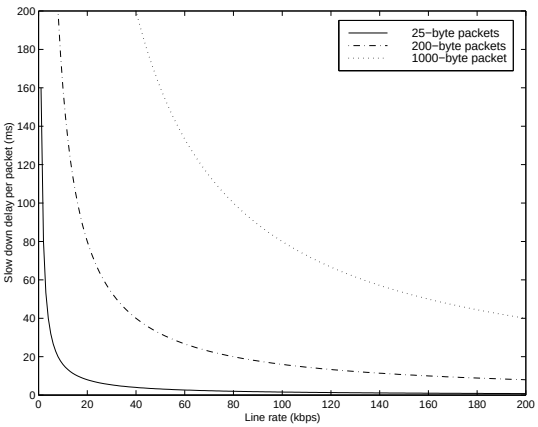


Figure 2. *The computed delay per packet versus requested line rate*

Taking into account the accuracy of sleeps, and observing Figure 2, we see that in standard Linux modeling of RLC is out of question, LLC can be modeled with meaningful results up to 20 kbps[2], and only the IP-level seems to be manageable for higher line rates. In practice, even IP-level modeling would give inaccurate results, because sleeps are always greater than requested and the accumulated delay would result in the line rate of the emulated link to be lower than requested.

In modeling a data link some amount of variation of delay per packet is acceptable, and sometimes even natural, because it is also present on the real link. However, the errors in individual sleeps should not accumulate, or otherwise the results would be biased.

Events for downlink and uplink channels of the Seawind simulation process are scheduled concurrently. Because of this an average sleep request would be half of that is given in Figure 2. Note also, that it only accounts for slow down sleeps, so if a process is interrupted during the sleep to process some event, for example background load packet arrival, and then goes to sleep again, the error may be much larger.

[2]We assume 1 kbps is 1000 bps, but 1"Kbyte is 1024 bytes.

3.2 Formalization of the problem

In this section we give a number of numerical parameters, that can be used in the comparison of different methods of accurate sleep. All sleep requests can be roughly divided into two groups. The first group consists of one-occurrence sleeps that are not dependent on each other. An example is a random delay modeling the effect of some rare event, for instance, a cell change. The accuracy of such sleeps is more difficult to improve, but on the other hand such sleeps tend to be rare and large in value, thus the relative error for such requests is small.

The second group consists of sleeps belonging to a single sleep thread or, in other words, a series of sleep requests. An example is emulation of a slow link, when a delay is done per packet of a data flow. Some difference between the requested and actual sleep time per one sleep in a thread is acceptable, as long as, on the average, the actual sleeps are the same as requested. This is sometimes called *error dumping* [5]. The value of individual requests and the length of the series is often not known in advance.

Let x_i be the requested sleep times belonging to the same series and let y_i be the actual times elapsed for the ith request, $i = 1, \ldots, n$ for some $n \in N$. We define the absolute sleep error as

$$a_i = y_i - x_i$$

and the relative sleep error as

$$r_i = \frac{y_i - x_i}{x_i} = \frac{a_i}{x_i} .$$

If Z_i, $i = 1, \ldots, n$ are random variables, we denote the sample mean as

$$\bar{Z} = \sum_{i=1}^{n} \frac{Z_i}{n}$$

and sample variance as

$$\sum_{i=1}^{n} \frac{(Z_i - \bar{Z})^2}{n-1} .$$

Naturally, we wish a_i and r_i to be constantly zero, that is equivalent to having zero sample sample mean and variance. We will use the sample

mean and variance of the absolute and relative error as a rough estimate of how good the suggested methods are. It is acceptable to have the small non-zero variances because they only reflect the deviation of individual sleep requests that are often present in the real system as well. However, the means should be kept as close to zero as possible because the indicated bias directly affects the final results.

The relative error shows how well a method approximates an area of the smaller sleep request values, because even a small absolute error there would result in a large relative error. On the other hand, the absolute error gives the way the method behaves "on average" and allows to estimate how large an error is introduced in the final results.

Table 1 gives a summary of error statistics for sleep using a `select()` system call on a standard PC Linux.

Table 1. *Statistics for different sleep techniques. Results are measured based on 1000 sleep requests are uniformly distributed in 0 ... 100 ms*

method	absolute error sample mean	absolute error sample variance	relative error sample mean	relative error sample variance
standard `select()`	14.50	10.49	0.85	4.39
interrupts RTC	0.00	0.00	0.00	0.00
interrupts HZ	0.41	0.25	0.04	0.01
slack variable	0.01	38.56	-0.09	0.27
busy waiting	0.02	0.11	0.00	0.00

3.3 Background of the problem

The standard Linux kernel sets the frequency of the timer interrupt to 100 Hz at boot time that corresponds to 10 ms interval between interrupts. When a process requests to be temporarily suspended and woken after some specified time, a timer structure is created and added to a list maintained by the kernel.

At each interrupt, the kernel increments a number of ticks by one. The interval length between timer interrupts is called a *jiffy*. Since the kernel checks for expired timers only when a timer interrupt occurs, the smallest meaningful sleep request time is one jiffy. In fact, the POSIX standard for `select` system call states that the process must sleep at least the time requested. To guarantee this, a kernel adds one jiffy to the requested sleep time in jiffies. That means the smallest sleep time in practice is two jiffies[3].

Fortunately in the modern Linux kernel `gettimeofday` provides nearly microsecond accuracy employing a time-stamp register (TSR) available on Pentium processors that is incremented on each clock cycle. Earlier kernel versions returned the time-of-day value updated only at a timer interrupt.

3.4 Possible solutions

Methods of solving the problem of accurate sleeps can be divided into three groups:

1. Using some mechanism to get finer clock resolution.
2. Compensating the difference in the next sleep request.
3. Busy waiting.

In the first group, the frequency of timer interrupts is increased either permanently or temporarily, and interrupts are handled either by the kernel or by the user process. In the second group, the requested sleep time is changed to reflect the error made in previous sleeps or to match the expected actual sleep time. In the third group, the accurate `gettimeofday()` call is used to actively wait until the requested time has elapsed.

Methods are then compared using the following evaluation criteria:

- high accuracy (small absolute and relative error),
- transparency for applications,
- load on the CPU,
- amount of modifications needed to the kernel,

[3]In kernel versions 2.2 and later the smallest sleep time is reduced to one jiffy.

3.5 Measurement specifications

3.5.1 Measurement model

Initially the following parameters were identified as possibly affecting the results:

- the pattern of sleep requests by the application,
- the overall system load,
- the amount of computation in the application,
- the length of the sleep series.

After consideration, a decision was made to use a long series of uniformly distributed in 0 ms to 100 ms requests on unloaded system. The pattern of requests is different for each application and thus difficult to generalize. The overall system load may have different effect depending on the priority of the real-time application. Computation time between sleep requests can be withdrawn from the sleep time requested and thus should not affect the results. The length of the sleep series was chosen of 1000 requests. This is longer than most sleep series in practice, but allows for better statistics.

The sleep request series was generated by a C-program using standard Linux `random()` call. The series was the same for all tested methods. A number of shell scripts and short programs in C-language were written to compute the relative and absolute error, sample mean and variance and to plot figures. All 1000 samples were used for statistics, but only the 100 first samples are shown in figures to keep the size of graphics files manageable.

3.5.2 Test environment

Performing tests required three different Linux kernels to be installed on a single machine. In Linux it is possible to keep multiple kernel boot image files and switch between then on a system boot. A convenient interface is achieved using (Linux Loader) LILO tools.

Software. Linux kernel 2.0.36, Computer Science Linux distribution (modified Slackware), gcc 2.7.2.3, libc5 library, ELF executables.

Hardware. Pentium II 450 MHz CPU, 128 MB RAM, 2 FUJITSU 4325 MB HDDs.

3.6 Methods of accurate sleep with kernel support

3.6.1 Counting RTC interrupts

Linux provides a driver to control the RTC chip, so the interrupt rate of the RTC can be set with a `ioctl()` calls and the process is informed of the interrupt occurrence using `read()` or `select()` system calls on the `/dev/rtc` device.

The support for RTC in the kernel is optional and can be activated when the kernel is compiled. At our department installation this option is disabled, and a sample kernel had to be compiled with RTC support enabled to perform tests.

Figure 3(b) shows that this method produces fairly accurate results. In fact, as can be seen from Table 1, all actual sleeps are exactly as requested when rounded to milliseconds. System tools indicated 0% CPU utilization when running the test process.

A negative side of this method is that it requires a replacement of the sleep routine in the applications. The Mowser library would need a major change to be able to use the RTC interface.

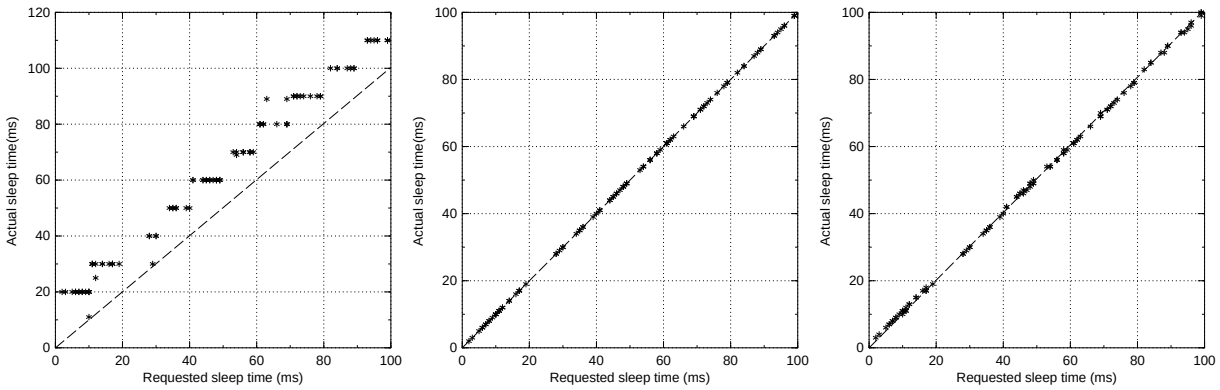
3.6.2 Increasing the interrupt frequency of the kernel

The frequency of timer interrupts, and thus accuracy of *select* call is affected by the value of the HZ constant in kernel sources. It is defined in the `include/asm-i386/param.h` file. The default value is 100, but it is possible to change within the range of the clock chip capabilities. Increasing the frequency of clock ticks has a negative impact in CPU overhead. As the Seawind system aims at approximately 1 ms resolution, the value of *HZ* of 1024 can be considered appropriate.

A sample kernel was compiled with this feature and measurements were run. Figure 3(c) and Table 1 show the results. The main advantage of this method is the complete transparency for applications.

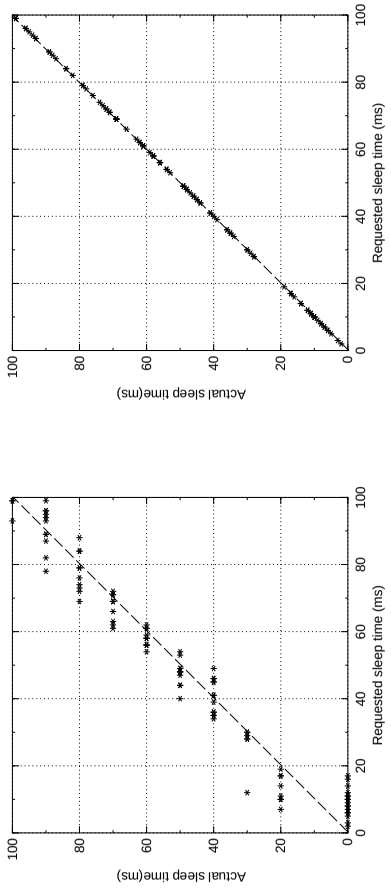
3.6.3 UTIME patch

UTIME is an extensive modification of the kernel that aims at providing accurate timing without putting an excess load on the system. It is done by increasing the frequency of the timer only temporarily, only when this is actually needed, because even if events are scheduled with microsecond



(a) Basic case (b) Counting RTC interrupts (c) Frequent kernel interrupts

Figure 3. *Performance of different sleep techniques. A dashed line shows the optimal behavior. Graphs contain desired vs. requested sleep time for the first 100 requests*



(d) Sleep with slack

(e) Busy waiting

Figure 3. (Continue) *Performance of different sleep techniques. A dashed line shows the optimal behavior. Graphs contain desired vs. requested sleep time for the first 100 requests*

resolution they are rarely scheduled every microsecond. Rather than interrupt CPU at the fixed rate, the timer chip is programmed to interrupt CPU at the time of the earliest scheduled event. This approach yields good results, and the achieved accuracy is up to 50 μs [3].

However, UTIME does a large modification of the kernel, and it can possibly have some negative side effects. It is not a part of the official kernel that was verified by hundreds of independent people. Another problem is practical usability: the required patch only installs on the certain kernel version (2.0.34) and is aimed at RedHat distribution. It might also be considered somewhat an overshot, because currently Seawind needs only 1 ms resolution. For these reasons UTIME was not tested, but perhaps it will be checked more closely in future.

3.7 Methods of accurate sleep without kernel support

3.7.1 Sleep with slack

The average accuracy of a sleep thread can be improved by measuring the actual sleep time of the current request and compensating the difference later with the next sleep request. The interface to the sleep routine is modified to pass two parameters to the function: the time requested for a sleep and a pointer to a variable containing slack from the previous sleep request. The programmer is responsible for separating sleep threads in the application, and assigning the slack variables to them. The sleep routine in C-code is given below. The slack variable can contain the positive or negative value, depending on whether the previous sleeps were shorter or longer than requested. The slack variable is updated to the value compensated in the sleep.

```
int sleep_with_slack(int sleep_ms, int *slack) {
  int slept;
  if (sleep_ms-*slack<=0) {
    *slack-=sleep_ms;
    return 0;
  }
  slept=ms_sleep(sleep_ms-*slack);
  *slack-=(sleep_ms-slept);
  return slept;
}
```

This method does not increase the accuracy of a single sleep call, of course. However, as can be seen from Figure 3(d), actual sleep times are evenly distributed around the requested time. Table 1 shows that the absolute error is very low, thus on average the actual sleeps are same as requested.

The best side of this method is that it can be used on unmodified kernels. It can be successfully combined with other methods that require kernel support to further increase the accuracy accounting for deviations in individual sleep requests.

3.7.2 Sleep with pre-compensation

It is easy to note that the sleep time provided by the unmodified `select()` tends to be larger than requested approximately by the constant component of 10 ms plus a variable part that varies from 0 to 9 ms depending on the least important digit.

In the method we call *sleep with pre-compensation* a requested sleep time is decreased by the value of the expected oversleep. This can be done inside the application or by modifying the sleep routine. For standard sleep with `select()` it decreased the errors, but not enough to make this method useful by itself.

It was interesting to check if pre-compensation would improve the performance of sleep with slack. In fact, the experiment has shown that there is no significant difference when pre-compensation is used. At first it was surprising, but later it was observed that the slack variable tends to stabilize at the value typically requested by pre-compensation.

3.7.3 Busy waiting

We mentioned in Section 3.3 that the `gettimeofday()` call provides nearly microsecond resolution in time. It is possible to wait for an exact time period by repeatedly calling `gettimeofday` until the requested time has elapsed.

However, this approach would not work for an event-driven application, as Seawind is. All event handlers must be kept short not to block the processing of other pending events, and busy waiting inside a handler is certainly unacceptable. A better solution is busy waiting through the Mowser dispatcher itself. This is possible because Mowser supports event

handlers of different priorities. In this way a handler `mev_later` is registered with a zero timeout and minimal priority. If there are any pending events, they will be processed first, and then a function for timer event is called. This function checks if the time of request has already elapsed, and if not, re-register the handler in the same way.

This method can be used when there is a single process per CPU. For a multi-process system only very short sleeps can be done in such a way, otherwise the sleeping process will use up all its CPU quota only for busy waiting and will be preempted. An advantage of the method is high accuracy.

Figure 3(e) and Table 1 show the results. For all but one request the error is zero. This single request well illustrates the shortcoming of this method, as the probable reason for it is a preemption of the process that has used up its CPU quota.

4 Other problems

In this section we outline miscellaneous issues that affect the accuracy of simulation results. All of them need closer consideration, which in turn requires benchmarking. Some of the problems were already experimented with, so possible solutions are also given.

4.1 Disk I/O

Seawind processes access the disk for reading configuration and writing log. All configuration related information is read before starting the experiment and thus should not be a problem.

In Seawind, an experiment consists of repetitions of basic tests (for example one TCP connection), so the log writing should not cause additional problems for short basic tests, because the log can be stored entirely in the main memory during each basic test and written to disk between basic tests.

For more intensive logging (when, for example, whole packets are logged) and longer tests the problem remains. A good method of real-time logging is to keep a number of buffers each of the size of a disk sector in the main memory, and asynchronously write a full buffer to disk while filling the other buffers [6]. The appropriate number of buffers should

be determined experimentally. Performance of asynchronous I/O under Linux needs closer consideration, because it is currently done without kernel support, but with a separate user thread per each request.

In general, providing a lightweight and predictable I/O is a fairly difficult task that requires close consideration and possibly replacement of some Linux kernel components [4].

4.2 CPU and memory performance

Even if on average occurring events require a small amount of time to process, situations are possible when several events are scheduled close to each other. For example, a bunch of background load users have arrived and need to be processed almost instantly. Some delay in dispatching events is inevitable in this case, but it is important to find out how large it is and how it can be accounted for.

The overall system performance can be of concern when the simulation model involves much computing. The Seawind code would need to be profiled and analyzed to remove the bottlenecks. In particular, data inside the simulation process often need to be copied without actually modifying them. In some cases this can be avoided by more careful programming.

4.3 Clock synchronization

The PC clock chip is accurate to 13 min per year at normal temperature and a fresh battery [7]. In smaller units, it is approximately 90 ms per hour. This is large enough to impose problems with the analysis of logging data, as logging is distributed. Some mechanism should be used to either find out the offset of each computer's clock or to synchronize them. A `timesync` script is available on all CS department Linux machines. It uses Network Time Protocol (NTP) to synchronize the clock on a computer with the clock of the time server.

4.4 Process scheduling

Standard Linux processes use SCHED_OTHER default universal scheduling policy, which aims at optimizing throughput rather than fulfilling requirements of real-time processes. If besides a simulation process, other active processes are present on the same computer, it is important that the

simulation process is given highest priority so that it cannot be preempted by other processes. Fortunately, Linux fulfills the POSIX requirements for soft-real time systems and provides two other scheduling policies for special time-critical applications that need precise control over the way in which runnable processes are selected for execution [6]. In order to determine the process that runs next, the Linux scheduler looks for the non-empty list with the highest static priority and takes the process at the head of this list. All non-real time processes run under the static priority of 0. In opposite, a real-time process can assign itself a static priority in the range 0 to 99. All scheduling is preemptive, if a process with a higher static priority gets ready to run, the current process will be preempted and returned into its wait list.

For real-time processes two scheduling policies are available, First In First Out (SCHED_FIFO) and Round Robin (SCHED_RR). SCHED_RR is a simple enhancement of SCHED_FIFO, the only difference is that in SCHED_RR each process is only allowed to run for a maximum time quantum. If a SCHED_RR process has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority. Assigning scheduling policies and priorities for processes of a given system can be done only based on the exact functionality of each process.

A computer system used as a platform for running experiments should be as bare-boned, as possible. In particular, the X server should not be used, but rather a single textual shell. Care should be taken to remove miscellaneous system processes and daemons that are not needed for the real-time processes, but are present on the normal Linux system, because any such process is a potential source for scheduling distortions for a real-time application.

4.5 Virtual memory paging

Virtual memory paging can cause unexpected delays in the execution of real-time processes. In the first place, paging should not happen at all during the run of an experiment, but in some cases (for example when log file is kept in the main memory) is possible. To prevent this problem from occurring, a `mlockall` system call should be used. It disables paging for all pages mapped into the address space of the calling process. This

includes the pages of the code, data and stack segment, as well as shared libraries, user space kernel data, shared memory and memory mapped files. All mapped pages are guaranteed to be resident in RAM when the `mlockall` system call returns successfully and they are guaranteed to stay in RAM until the pages are unlocked again by `munlockall` or until the process terminates.

5 Conclusion

Miscellaneous technical issues are shown to be crucial for a sound implementation of a real-time network simulator. The most important problem is to provide a simulation process with an accurate delay mechanism that does not interfere with other processes. Several methods were evaluated and their usability was discussed.

The excellent accuracy in sleeps can be achieved by *counting RTC interrupts* at the expense of modifications to applications and possibly to the kernel. *Busy waiting* provides the high accuracy as well, however, periodic distortions are possible due to intensive CPU utilization. This method requires a modification to applications, but not to the kernel. *Increasing frequency of kernel interrupts (HZ)* gives high accuracy with complete transparency for applications, but does require a modified kernel. Finally, *sleep with slack* allows for good accuracy on average for longer sequences of sleep requests on standard kernel with no overhead. However the programmer has to identify sleep threads in the application.

While no method was found to satisfy all the criteria, strong and weak sides of each method were identified to make an appropriate choice for particular purpose of a simulation study. The other important problems were briefly discussed, but more elaborate research is a subject to further work.

Acknowledgments

I express my deep gratitude to prof. Timo Alanko for reviewing the paper, for invaluable help and support, and to Dr. Iouri A. Bogoiavlenski for advice and encouragement. I thank Aki Laukkanen for useful comments on the paper.

References

[1] H. Helin, A. Gurtov, *Mowgli Communication Services: Mowser library.* Technical report, University of Helsinki, December 1999.

[2] A. Law, D. Kelton, *Simulation modeling & analysis.* McGraw-Hill series in industrial engineering and management science, second edition, 1991.

[3] B. Srinivasan, *A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software.* Master of Science thesis, University of Kansas, 1998.

[4] R. Hill, *Improving Linux Real-Time Support: Scheduling, I/O Subsystem, and Network Quality of Service Integration.* Master of Science thesis, University of Kansas, 1998.

[5] K. Atkinson, *An Introduction to Numerical Analysis.* John Wiley & Sons, 1978.

[6] B. Gallmeister, *POSIX.4: Programming for the real world.* O'Reilly & Associates, 1995.

[7] H. Messmer, *The Indispensable PC Hardware Book.* Addison-Wesley, third edition, 1997.

[8] A. Gurtov, *Technical Issues of Real-Time Simulation in Linux.* Bachelor Thesis, University of Petrozavodsk, 1999.

[9] M. Kojo, K. Raatikainen, M. Liljeberg, J. Kiiskinen, and T. Alanko, *An Efficient Transport Service for Slow Wireless Telephone Links.* IEEE Journal on Selected Areas in Communications, vol. 7, no 15, pp. 1337–1347, 1997.