Technical Issues of Real-Time Network Simulation on Linux

B.Sc. thesis

Andrei Gurtov

Petrozavodsk State University
Faculty of Mathematics
Department of Computer Science

Technical Issues of Real-Time Network Simulation on Linux

Andrei Gurtov

B.Sc. Thesis

Petrozavodsk State University
Faculty of Mathematics
Department of Computer Science
29th of May, 1999
31 pages

Abstract

The real-time network simulation requires dealing with miscellaneous technical problems to achieve a correct and timely execution. The ignorance of 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 goals of a simulation study.

Computing Reviews Classification: I.6, D.4

Keywords: Real-time simulation, Linux, network modeling.

Contents

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 behavior of Internet protocols over a real data link or network is often costly or, if it 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 a 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 have to compete with other processes for system resources.

For example, consider Figure 1. It presents performance results from the first version of 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 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 of 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 of related work is concentrated only on achieving the highest possible accuracy, but ignoring practical factors that are sometimes decisive for usage of a method. In this paper, we take into consideration such issues as the amount of modifications needed in the kernel, transparency of a method for applications, and maintainability of the computer system.

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 goals of a simulation study. Other problems are outlined and possible solutions to them are suggested, but an extensive evaluation is a subject of future work.

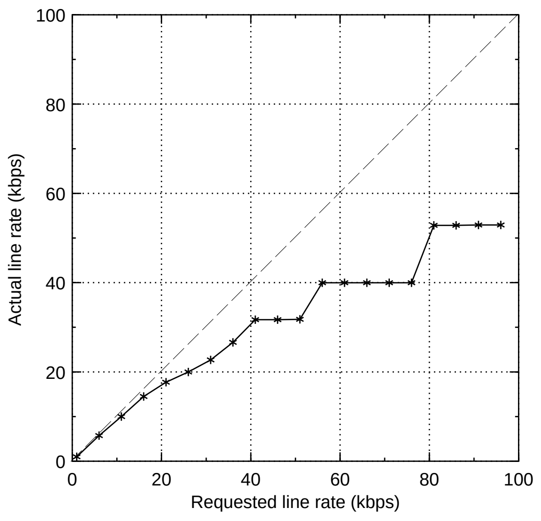


Figure 1: Actual versus requested line rate. Measured with WINES simulator using 100-byte packets. Sleeps are performed using a standard Linux system call.

[1]We use PC to refer to any of personal computers based on i386 and its successors

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. It may be classified as a real-time distributed functional simulator [2]. The simulation system consists of several simulation 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 an 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 future.

Figure 2 shows the Seawind components and interfaces between them. 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.

Figure 3 shows the internal architecture of the simulation process. The heart of it is a simulation kernel that includes two uni-directional data channels referred to as **uplink** and **downlink**. Auxiliary functions (configuration, logging, random number generation from a distribution) are handled by modules external to the simulation kernel.

The approximate size of the whole system in the first phase is approximately eleven thousand lines of code, that includes the simulation process, GUI and control components. In the second phase, when additional features are implemented,

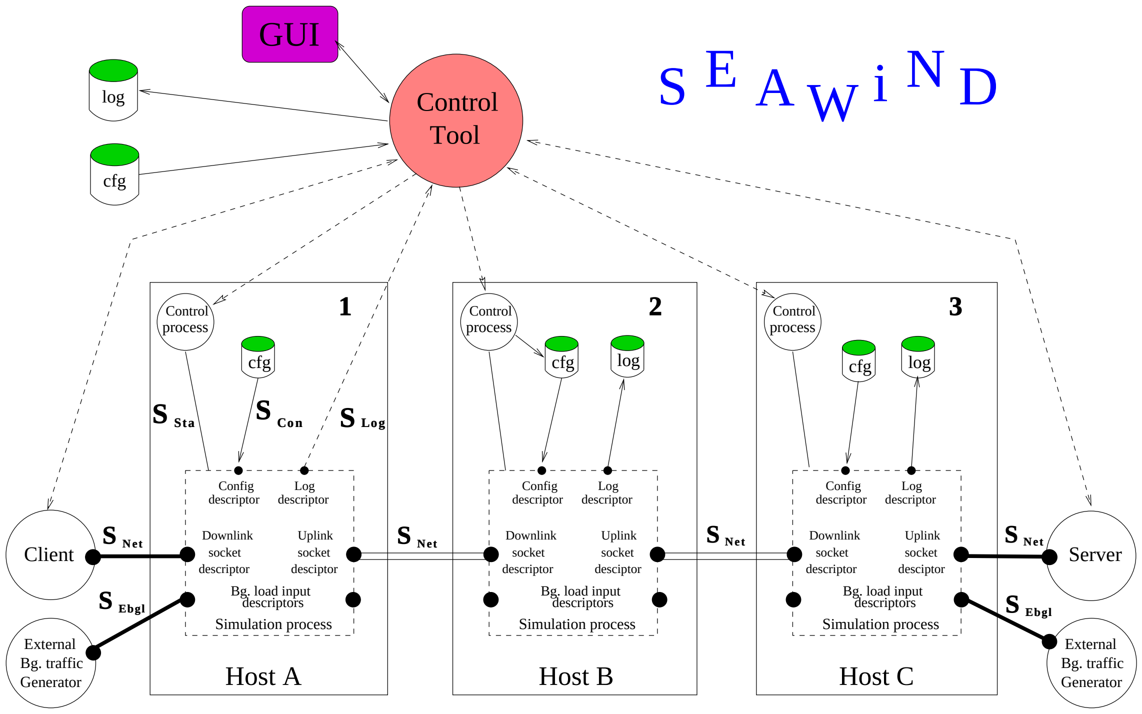


Figure 2: Seawind simulation system.

the code size can double.

Currently the maximum envisaged number of simulation processes is four. As a rule, every simulation process should be run at a separate PC. It is expected that the Seawind simulator will be used outside of our department as well. These facts imply that it is not wise to demand the usage of a modified Linux kernel for all experiments. In this paper we outline the cases in which the kernel modification is a must, and cases there required accuracy can be achieved by suggested user software methods.

3 The problem of an accurate sleep time

3.1 Definition of the problem

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

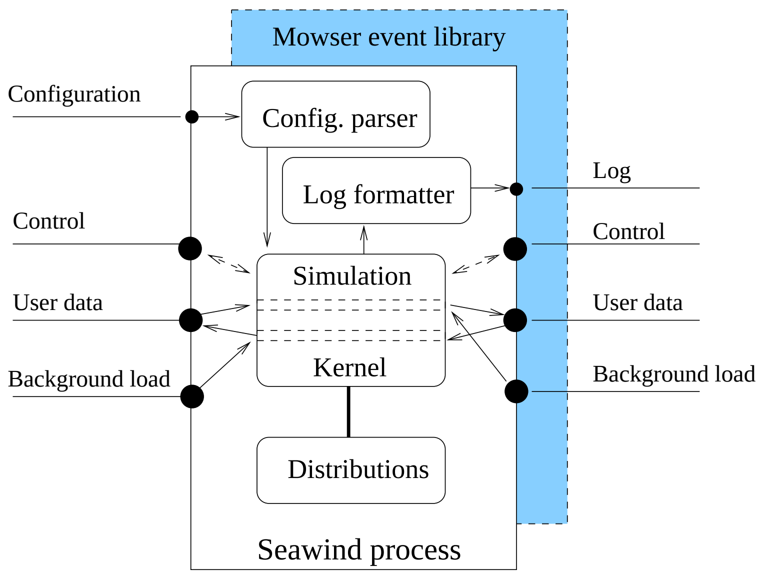


Figure 3: Simulation process.

10 ms more than requested. In the later sections we will see reasons for such coarse behavior, but first we consider implications of these facts to our real-time network simulator.

Figure 4 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 of 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). Note, that in this paper we assume 1 kbps is 1000 bps, but 1 kbyte is 1024 bytes.

Taking into account the accuracy of sleeps, and observing Figure 4 we see that on standard Linux modeling on RLC is out of question, LLC can be modeled with meaningful results up to 20 kbps, and only 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 accumulated delay would result in the line rate of emulated link to be lower than requested.

In modeling of a data link some amount of variation of delay per packet is acceptable, and sometimes even natural, because it also presents on the real link.

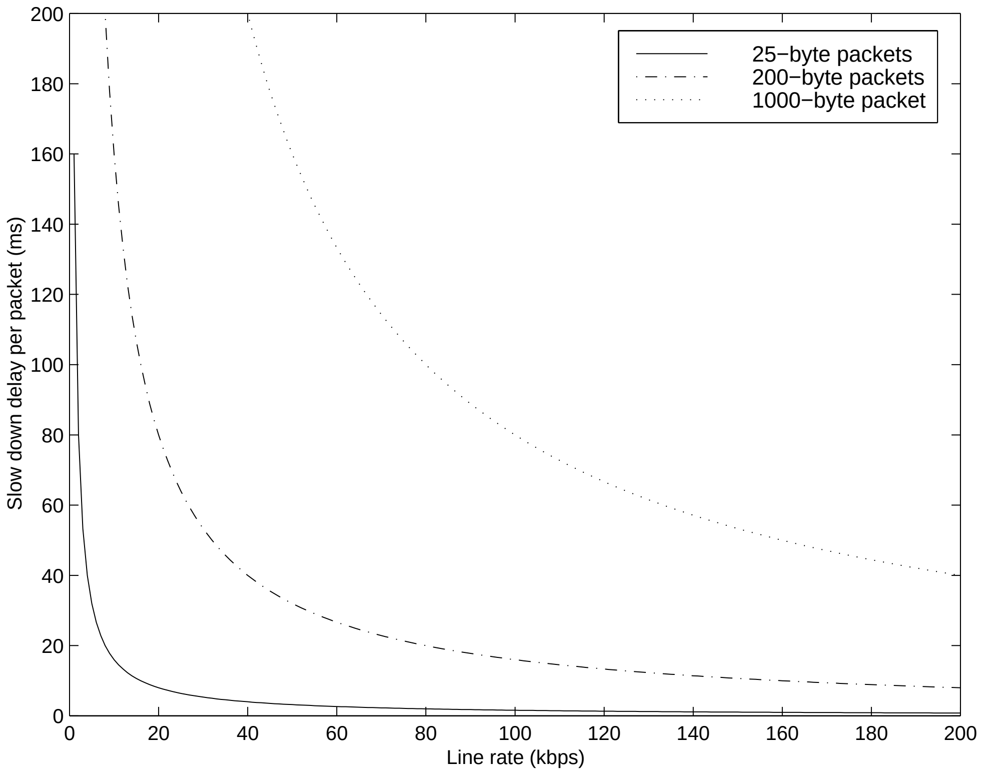


Figure 4: The computed delay per packet versus requested line rate.

However, the errors in individual sleeps should not accumulate, or otherwise the results would be biased.

Events for downlink and uplink channels of Seawind simulation process are scheduled concurrently. Because of this an average sleep request would be half of that Figure 4 gives. 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 can be twice as large.

3.2 Formalization of the problem

In this section we give a number of numerical parameters, that can be used in 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. As we see later, accuracy of such sleeps is more difficult to improve, but on the other hand such sleeps tend to be large in value, thus the relative error for such requests is small.

The second group consists of sleeps belonging to 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 is same 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 ith request, $i = 0..n$ 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 $Z_i, i = 1..n$ are random variables, we denote the sample mean as

$$\hat{E}(Z_i) = \sum_{i=0}^{n} \frac{Z_i}{n}$$

and sample variance as

$$\widehat{Var}(Z_i) = \sum_{i=0}^{n} \frac{(z_i - \hat{E}(Z_i))^2}{n-1}$$

Now ae_i and re_i are random variables and we have

$$AESM = \hat{E}(a_i)$$

absolute error sample mean and

$$AESV = \widehat{Var}(a_i)$$

Table 1: Statistics for sleeps using a standard Linux system call. N is number of requests, AE and RE are absolute and relative error, SM and SV are sample mean and sample variance.

N	AE SM	AE SV	RE SM	RE SV
1000	14.50	10.49	0.85	4.39

absolute error sample variance.

Correspondingly

$$RESM = \hat{E}(r_i)$$

relative error sample mean and

$$RESV = \widehat{Var}(r_i)$$

relative error sample variance.

Naturally, we wish to have a_i and r_i to be constantly zero, that is equivalent to having $AESM = AESV = RESM = RESV = 0$. We will use $AESM, AESV, RESM, RESV$ as a rough estimate of how good the suggested methods are. It is acceptable to have the non-zero variances because they only reflects the deviation of individual sleep requests that often presents 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 (RESM, RESV) 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 (AESM, AESV) gives how does the method behave "on average" and allows to estimate how large error is introduced in the final results. For a given application the choice between methods with either smaller AE or smaller RE should be made based on the pattern of sleep requests: if the application tends to request smaller values that are close to the sleep resolution it is better to use the method giving the lower RE, otherwise a method with the lower AE.

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

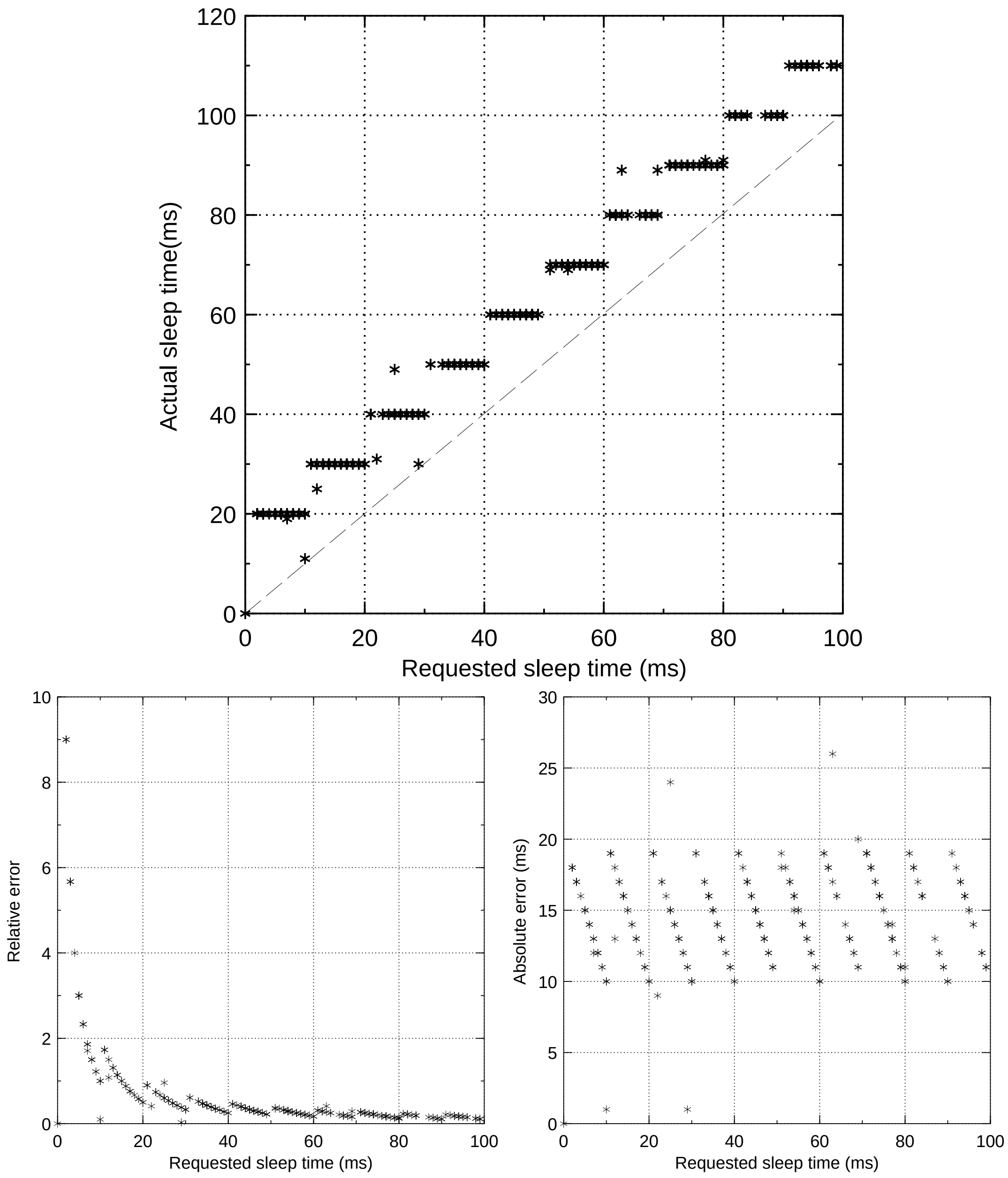


Figure 5: Performance of sleeps using a standard Linux system call. A dashed line shows the desired behavior.

3.3 Background of the problem

In this section we give an overview of the timer chip and explain the reasons for coarse timing.

Nearly every PC has a MC146818 (or clone) chip providing realtime clock as well as CMOS RAM. The realtime clock is independent on CPU and all other chips and it is powered by a battery when a PC is off, so that the clock can go on. MC146818 can be programmed to generate periodic interrupts. The chip has 4 bits of an internal status register to determine the rate of interrupts. The values can be in the range between 0011 and 1111. The frequency of interrupts of interrupts is then

$$f = 65536/2^{rate}$$

and lays between 8192 Hz (cycle time 122 μs) and 2 Hz (500 ms) [7].

The interrupt from MC146818 is reported on IRQ8. The frequency of interrupts can be changed with first writing the identification of the register containing rate bits (0ah) to the port number of MC146818's address register (70h) and then setting the corresponding bits of port (71h) of MC146818's data register to desired value.

Standard Linux kernel sets 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 waken after some specified time, a timer structure is created and added to a list maintained by the kernel. Simplified timer structure:

```
struct timer {
  (void *)func(unsigned long);
  unsigned long data;
  unsigned long expires;
  struct timer *next;
  struct timer *prev;
}
```

A field *expires* gives when the function *func* should be called passing *data* as a parameter. The timer list is double-linked and maintained in ascending time order.

At each interrupt, the kernel increments a number of ticks passed since it was started by one. The interval length between timer interrupts is called a *jiffy*. A framework of interrupt handler:

```
void handle_timer_interrupt() {
  jiffies++;
  check_timer_list();
  do_accounting_and_scheduling();
}
```

In this handler after updating the *jiffies* counter, the kernel calls *check_timer_list()* routine that check for pending timer events and process them as necessary. Finally, in *do_accounting_and_scheduling* routine kernel accounts the processes for CPU usage and possibly scheduling a new process.

```
void check_timer_list() {
  struct timer* timer_ptr=timer_head;

  while(timer_ptr!=NULL && timer_ptr->expires < jiffies ) {
    (*time_ptr->fn)(timer_ptr->data);
     timer_ptr=timer_ptr->next;
     remove(timer_ptr);
  }
}
```

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.

Fortunately in modern Linux kernel *gettimeofday* provides nearly microsecond accuracy employing 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 to solve 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 of 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 AESM, AESV, RESM, RESV),
- transparency for applications,
- load on the CPU,
- amount of modifications needed to the kernel,
- maintainability of the application and operating system.

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 should not have large effect, because the real-time application is supposed to be run with a higher priority than other applications. Computation time inside the loop should be withdrawn from the sleep request and thus does not affect the results. The length of the sleep series was chosen of 1000 requests. This is longer than most of sleep series in practise, but allows for better statistics.

Here is a code fragment used to generate the requests and output the (requested, slept) pairs. Note, that random generator is intentionally not initialized, so all methods are tested using the same sequence of requests.

```
for (i=0;i<1000;i++) {
    wanted_sleep_time=(double)random()/RAND_MAX*100;
    slept=sleep_function(wanted_sleep_time);
    printf("%d %d\n", wanted_sleep_time, slept);
}
```

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 300 first samples are shown on figures to keep the size of graphics files manageable.

3.5.2 Test environment

Hardware. Pentium II 450 MHz (450.56 bogomips)CPU, 128 MB RAM, 2 FUJITSU 4325 MB HDDs, 3Com 3C905B 100bTX Ethernet.

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

3.5.3 Using several Linux kernels

Performing tests required to have 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. As an example, we give a slightly edited *lilo.conf* file from a computer used for tests:

```
# LILO configuration file
#
# Start LILO global section
boot = /dev/hda3
delay = 50
# End LILO global section
# Linux bootable partition config begins
image = /vmlinuz
  root = /dev/hda3
  label = Linux
# Linux bootable partition config ends
# Linux bootable partition config begins
image = /vmlinuz_RTC_HZ_1024
  root = /dev/hda3
  label = LinuxHZ
# Linux bootable partition config ends
# Linux bootable partition config begins
image = /vmlinuz_RTC
  root = /dev/hda3
  label = LinuxRTC
# Linux bootable partition config ends
```

In principle, it is possible to supply a specially modified kernel with the simulation software. An installation and removal of this kernel can be done with a shell script. When a computer is used for normal purposes, a standard kernel should be selected from the boot menu, and when it is used for simulation, a special purpose kernel is selected.

3.6 Methods of accurate sleep with kernel support

3.6.1 Counting RTC interrupts

In Section 3.3 a port-based interface to program RTC chip was described. Fortunately, Linux provides a driver to handle this lower-level routine, so the interrupt rate of the RTC can be set with a `ioctl()` calls and the process is informed of the interrupt occurrence with 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.

A C-language code for sleep routine is given below. For simplicity error handling and code to ensure correct functioning with concurrent usage is omitted.

```
#define FREQ 1024
int fd;

int
rtc_sleep (unsigned int msec) {
  int retval;
  unsigned long data;
  int irqcount=0;

  if (msec==0) return 0;

  /* Set frequency of interrupts */
  retval = ioctl(fd, RTC_IRQP_SET, FREQ);

  /* Enable periodic interrupts */
  retval = ioctl(fd, RTC_PIE_ON, 0);

  /* Count interrupts */
  while (1) {
    retval = read(fd, &data, sizeof(unsigned long));
```

Table 2: Statistics for sleeps using RTC driver. N is number of requests, AE and RE are absolute and relative error, SM and SV are sample mean and sample variance.

N	AE SM	AE SV	RE SM	RE SV
1000	0.00	0.00	0.00	0.00

```
    irqcount+=data/256;
    if ((double)irqcount*1000/FREQ > msec) break;
  }

  /* Disable periodic interrupts */
  retval = ioctl(fd, RTC_PIE_OFF, 0);

  return (double)irqcount*1000/FREQ;
}

int main() {
 int i, retval;
 int wanted_sleep_time;
 int slept;

 fd = open ("/dev/rtc", O_RDONLY);

 for (i=0;i<1000;i++) {
    wanted_sleep_time=(double)random()/RAND_MAX*100;
    slept=rtc_sleep(wanted_sleep_time);
   printf("%d %d\n", wanted_sleep_time, slept);
 }
}
```

Figure 6 shows that this method produces fairly accurate results. In fact, as can be seen from Table 2, all actual sleeps their exactly as requested when rounded to milliseconds. System tools indicated 0 % CPU utilization when running the test process.

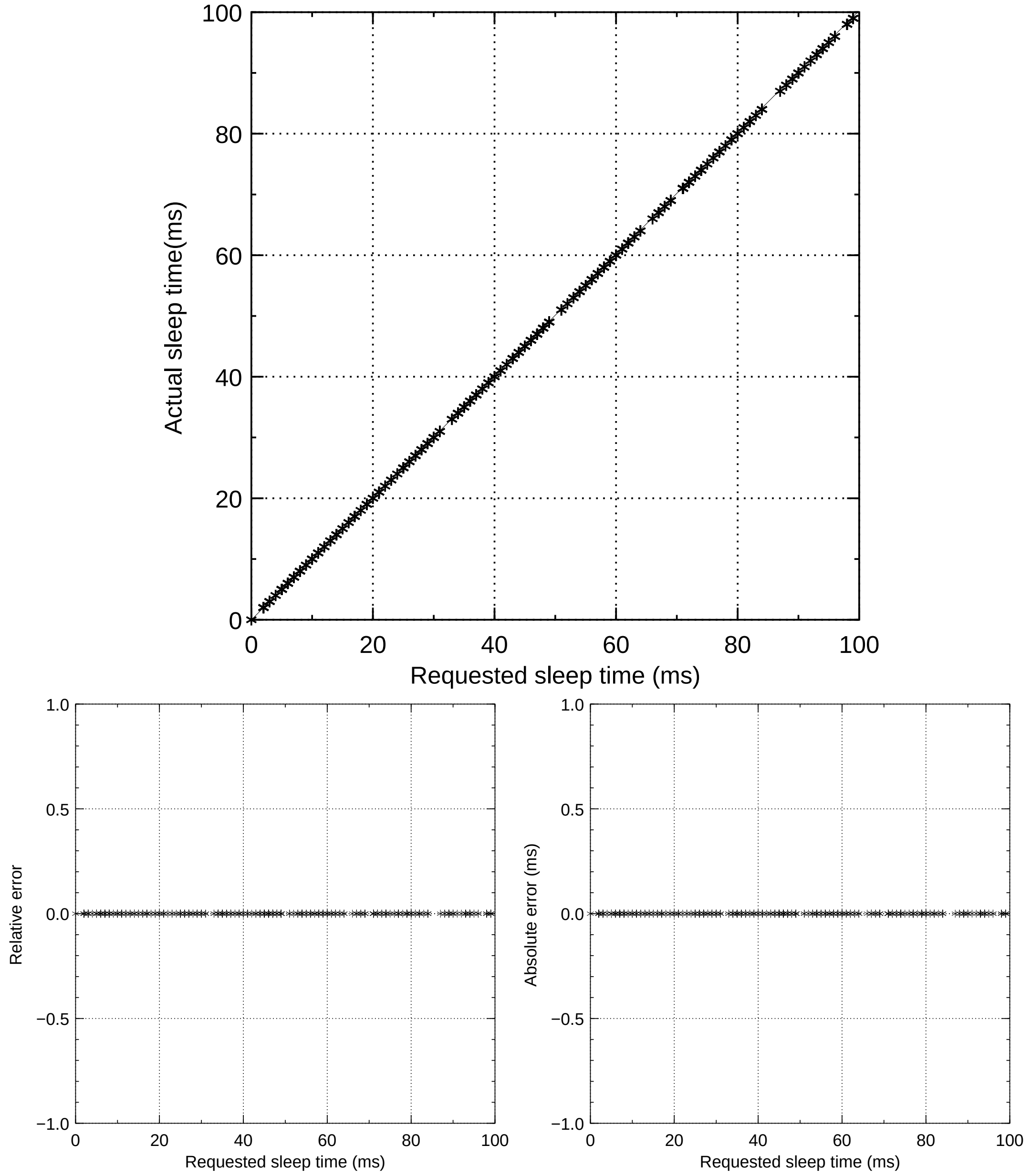


Figure 6: Performance of sleeps counting RTC interrupts. A line shows the desired behavior.

Table 3: Statistics for sleeps when frequency of kernel interrupts is increased. N is number of requests, AE and RE are absolute and relative error, SM and SV are sample mean and sample variance.

N	AE SM	AE SV	RE SM	RE SV
1000	0.41	0.25	0.04	0.01

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 able to use RTC interface.

3.6.2 Increasing interrupt frequency of the kernel.

The frequency of timer interrupts, and thus accuracy of *select* call is affected by the value of HZ constant in kernel sources. It is defined in *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 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 7 and Table 3 show the results.

The behavior, best observable from absolute and relative error graphs, has a simple explanation. As was mentioned in Section 3.3, Linux adds one jiffy to any sleep request. Because jiffy in this case is approximately 0.9766 ms, the absolute error is either rounded to 0 or to 1 ms. This can be fixed by using jiffy of 1 ms and requesting to sleep each time 1 ms less, but it will harm the main advantage of this method, 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 timer only temporarily, only when this is actually needed, because even if events are scheduled with microsecond resolution, they are rarely scheduled

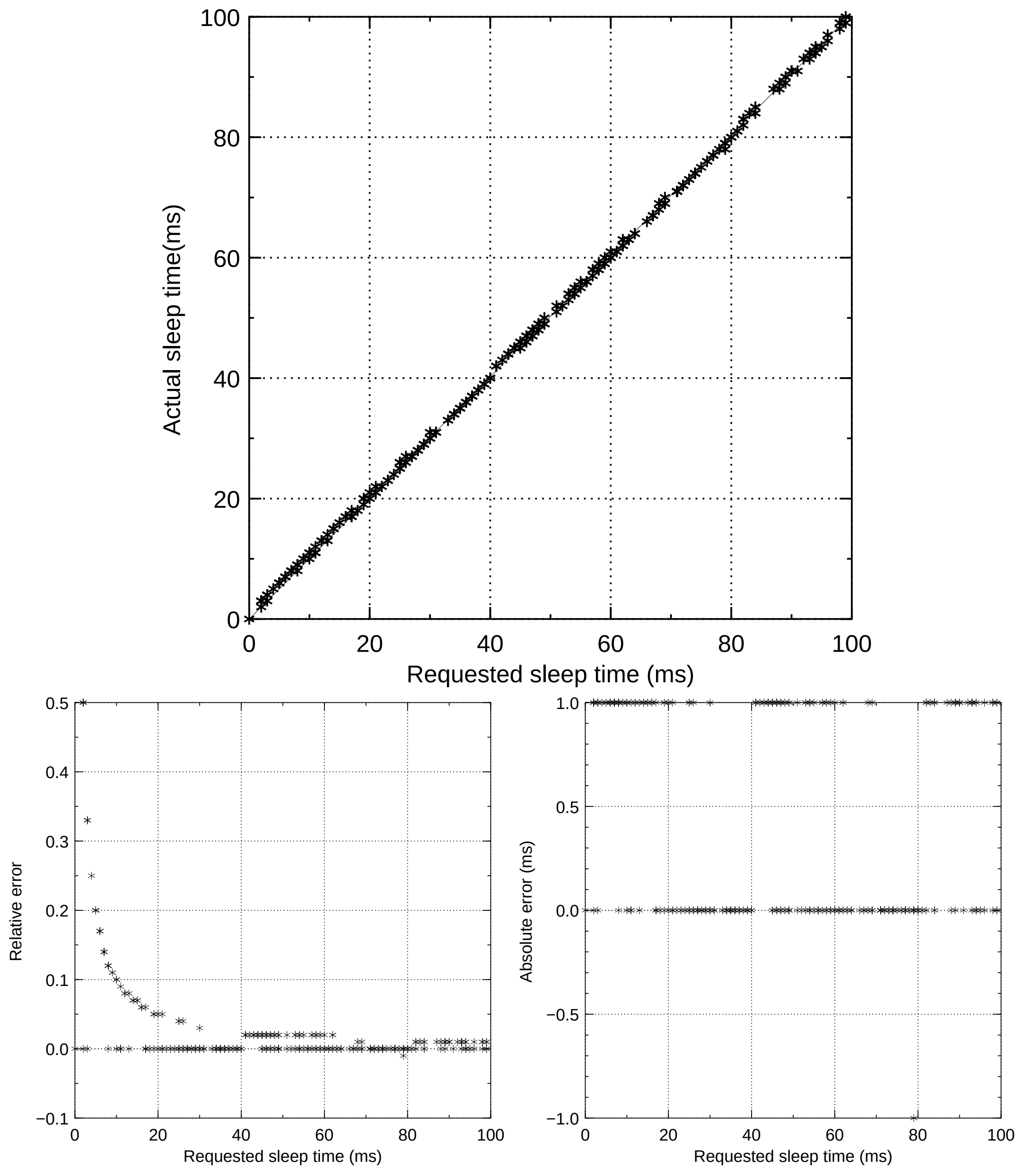


Figure 7: Performance of sleeps when frequency of kernel interrupts is increased. A line shows the desired behavior.

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 is very reliable because it was read and verified by thousands of independent people. In contrast, the UTIME code was probably checked only by a few programmers and some bugs are known but not fixed. Another problem is maintainability: the required patch only installs on the certain kernel version (2.0.34) and is aimed at RedHat distribution. Also it might be considered somewhat an overshot, because currently Seawind needs only 1 ms resolution. For these reasons UTIME was not tested, but perhaps it should 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. Unfortunately, it is not possible to do this without modifying the interface to the sleep routine, because the slack should be kept per sleep thread, and a pointer to this variable need to be passed to the routine on each call.

The interface to the sleep routine is modified to pass two parameters to the function: the time requested for a sleep and pointer to a variable containing slack from the previous sleep request (it can be positive or negative).

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 if 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) {
```

Table 4: Statistics for sleeps counting slack from previous requests. N is number of requests, AE and RE are absolute and relative error, SM and SV are sample mean and sample variance.

N	AE SM	AE SV	RE SM	RE SV
1000	0.01	38.56	-0.09	0.27

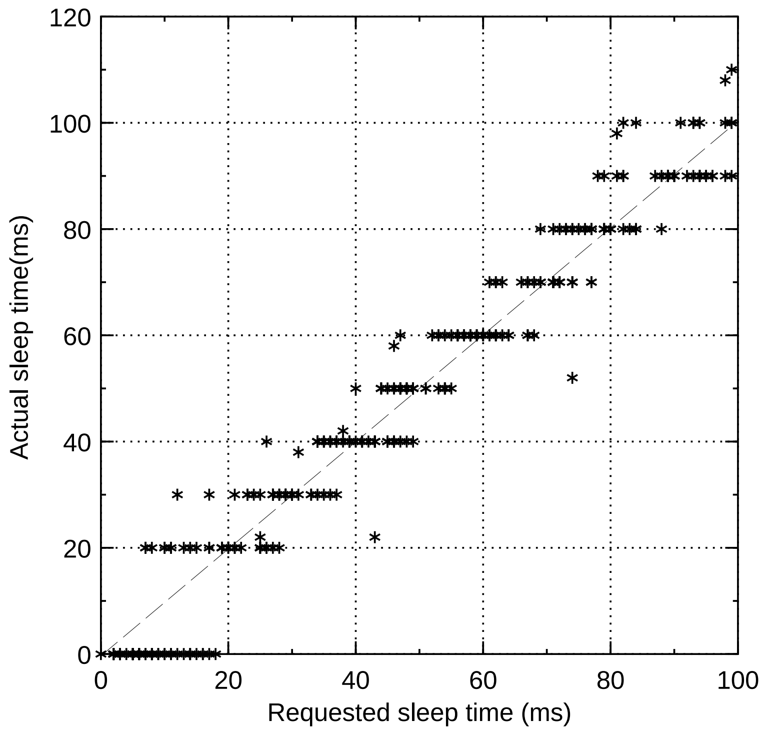
```
  int slept;

  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 8, the absolute error is evenly distributed around the zero, and the relative error is smaller than for standard sleeps in Figure 5. Table 4 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 is 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 easily noticed that the sleep time provided by the unmodified select tends to be larger than requested approximately by the constant component of 10 ms plus the a variable part that varies from 0 to 9 ms depending on the least-important digit.



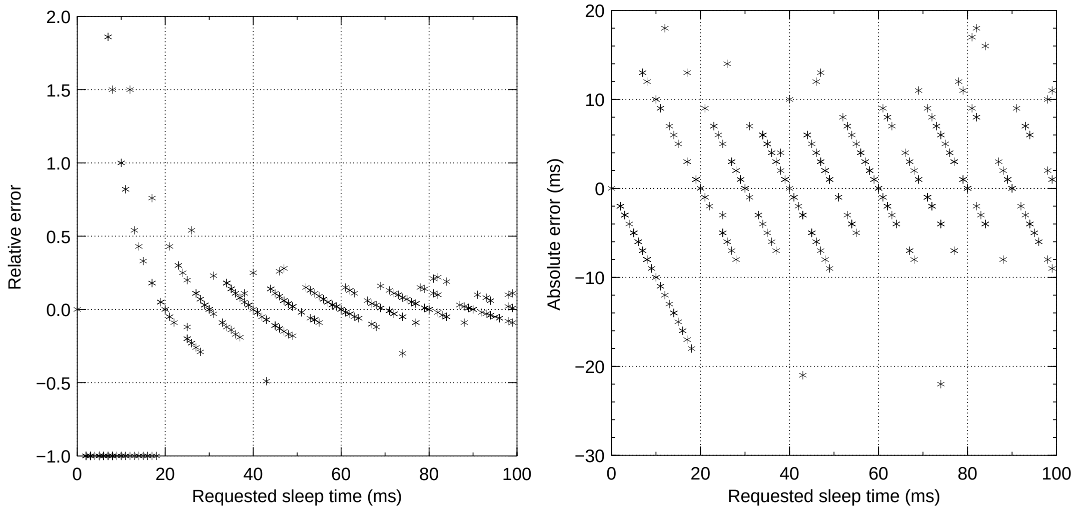


Figure 8: Performance for sleeps with slack. A dashed line shows the desired behavior.

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 then pre-compensation is used. At first it was surprising, but later it was observed that 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 *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. A C-code sleep routine is given below:

```
int
busy_wait(int msec) {
  struct timeval tv1,tv2;

  gettimeofday(&tv1,NULL);
  do {
     gettimeofday(&tv2,NULL);
  } while(ms_between(tv1,tv2)<msec);

  return ms_between(tv1,tv2);
}
```

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

Table 5: Statistics for sleeps with busy waiting. N is number of requests, AE and RE are absolute and relative error, SM and SV are sample mean and sample variance.

N	AE SM	AE SV	RE SM	RE SV
1000	0.02	0.11	0.00	0.00

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 multi-process system only very short sleeps can be done in such 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 9 and Table 5 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 waiting process.

4 Other problems

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

4.1 Disk I/O

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

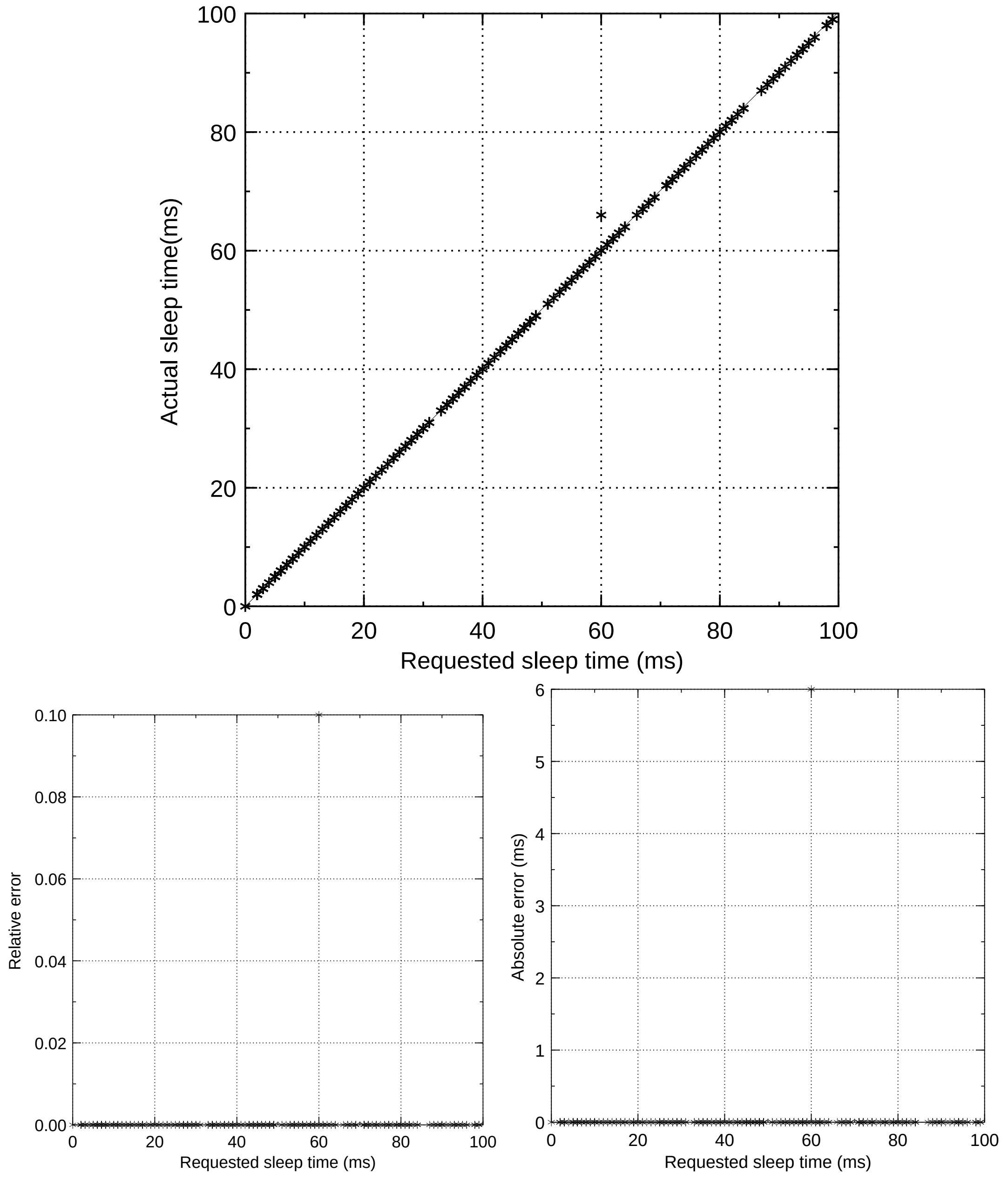


Figure 9: Performance of sleeps with busy waiting. A line shows the desired behavior.

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 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 is it and how can it be accounted for.

Overall system performance can be of concern with when simulation model involves much computing. The Seawind code would need to be profiled and analyses 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 analysis of logging data, as logging is distributed. Some mechanism should be used to either find out 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 clock on calling machine with the clock of the time server.

4.4 Process scheduling

Standard Linux processes use *SCHED_OTHER* default universal scheduling policy, that 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 the highest priority 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 of scheduling policies and priorities for processes of a given system can be done only based on exact functionality of each process.

A computer system used as a platform for running experiments should be as bare-bone, as possible. In particular, 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 delay in execution of real-time processes. In the first place, paging should not happen at all during run of experiment, but in some cases (for example 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.

4.6 Network buffers

Problem with network buffers is specific for network emulation when TCP flow control is used as a resort to stop sender from flooding SP with packets. In this case the sender application can also get a feeling of a slow link.

Default TCP send and receive buffers in Linux are 64 kbytes, so up to 128 kbytes can be filled by the sender almost immediately before the application will get blocked by TCP flow control. In reality, on 9600 kbps link, it should take approximately 110 sec.

A solution is to set the TCP send and receive buffers to a smaller value using *setsockopt* system call. This call will always succeed, but actually set value differs among kernel's versions. For example, for 2.0.35 SO_SNDBF and SO_RCVBUF are set to the maximum of requested value and 256 bytes, but on 2.2.7 SO_SNDBUF is set to the maximum of twice(!) the requested value and 2048 bytes and SO_RCVBUF to twice the requested value and 256 bytes.

The amount of buffered data should estimated for each experiment and caused error taken into account when evaluating the results.

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. Table 6 shows a summary of methods properties. 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 system configuration.

The other important problems were briefly discussed, but more elaborate research is a subject of further work.

Acknowledgments

I am grateful to Timo Alanko for reviewing the paper and to Markku Kojo for fruitful discussions on the topic.

Table 6: Summary of sleep methods

Method	**Accuracy**	**Application and Mowser library changes**	**CPU overhead**	**Kernel changes**	**Maintainability**
Counting RTC interrupts	Excellent	Sleep routine modification. Major modification of Mowser library	Negligible	RTC driver is optionally included in the kernel at compilation. Kernel with RTC can be well used as a normal PC	Department of CS computers do not have RTC driver installed. It would either be included in all machines at next kernel update, or a special kernel should be used for those machines during experiments
Increasing frequency of kernel interrupts	Good	No modifications required	$\approx$ 2%, acceptable in multi-process environment	A parameter in one of the header files is changed	Normal applications work with the modified kernel, although replacing the kernel permanently is unacceptable
Sleeping with slack	Satisfactory	Applications need to be modified to separate sleep threads. Interface to sleep function need to be extended to include slack variable	Negligible	Not required	Simulation can be run on a normal computer, thus there is no maintenance overhead
Busy waiting	Excellent	Sleep routine modification. Changes in code for applications using Mowser library	100 %, unacceptable in multi-process environment	Not required	Simulation can be run on a normal computer, thus there is no maintenance overhead

References

[1] H. Helin, "Mowgli Communication Services: Mowser library", Technical report, University of Helsinki, September 1998.

[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.