Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
Matthew Wilson, *rPath*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
Eugene Teo, *Red Hat, Inc.*
Kyle McMartin, *Red Hat, Inc.*
Jake Edge, *LWN.net*
Robyn Bergeron
Dave Boutcher, *IBM*
Mats Wichmann, *Intel*

Application Testing under Realtime Linux

Luis Claudio R. Gonçalves
Red Hat Inc.
lgoncalv@redhat.com

Arnaldo Carvalho de Melo
Red Hat Inc.
acme@redhat.com

Abstract

In the process of validating the realtime kernel and validating third-party applications under this kernel, it was necessary to build a set of small tools to understand different behavior presented by applications and the kernel itself.

We have identified several practices and common mistakes that could be harmful to performance and determinism in a Linux RT environment. We used systemtap and other tools to identify these problems and in some cases, to fix them or test alternatives without touching the application code.

1 Introduction

This paper talks about experiments conducted on systems with PREEMPT_RT Realtime enabled kernels. The main features of the PREEMPT_RT patch were already described in papers [1] and the Project wiki page [2].

The main goal of PREEMPT_RT is to offer determinism, predictability to the highest priority tasks in the system. The project is moving in a fast pace, and most of its mature features have already been merged upstream.

In the search for determinism, several well established entities had been touched and changed. Examples are:

- sleeping spin_locks – in the PREEMPT_RT patch most spin_locks were converted to rt_mutexes and can sleep. This leads to better kernel preemption capabilities.
- threaded interrupts – Interrupt Service Routines are now threaded. This prevents, for instance, lower priority tasks doing heavy I/O activity from creating latencies in higher priority tasks.
- threaded softirqs – each softirq has its thread and so the system administrator has the ability to change their priorities in order to favor the ones more important for his application.
- priority inheritance – avoid priority inversion by boosting the priority of a lower priority thread to the priority of the thread waiting for the resource, if higher. This is possible, in part, because rt_mutexes have the concept of ownership.

Sleeping spin_locks and threaded IRQs are bound together, because it is necessary to have a process context in order to sleep. This way, interrupt processing had to be delegated to specialized threads. Sleeping spin_locks are also building blocks for priority inheritance. These features are the foundations on which PREEMPT_RT is built.

Realtime is all about determinism, predictable behavior. There are certain practices in application development that may hurt these premises and lead a system to present a Byzantine behavior.

Along with determinism comes extra flexibility, as entities like interrupt handlers can now be prioritized, allowing the user to tune the environment to better server particular workloads. User applications can have priorities higher than any IRQ or kernel thread. Of course, application errors in such a high priority can lead to problems. A busy loop could starve disk IRQ or any other subsystem.

2 About the Tests

Most of the examples presented here are based in experience acquired while testing customer applications. Most of the tests happened under the “It runs slower on RT!” pressure. That alone was enough motivation to find the root cause of the observed behavior.

On one hand, the objective of Realtime is determinism and sometimes the cost of determinism is a negative impact on performance—other areas such as High Availability suffer from the same problem. On the other hand, the hit on performance should be as light as possible and Realtime, in fact, can improve performance in several scenarios.

That said, there are four main possibilities for the performance penalties observed:

- Problem in the kernel – a simple example is the case where a user application was aborting due to system load reaching the defined limit in the test, that was caused by a bug in the kernel system load calculation routine;
- Problem in the application – user application was multithreaded and not all the threads were running in the same priorities, or at least in reasonable priorities, causing the lowest priority thread to starve;
- Problem in both – race condition in user application that was more likely to be reached in Realtime, due to system architecture, triggered a kernel bug in a code path not usually exercised;
- Incorrect comparison – comparing results from running the test application in tuned environment versus running the test application in a system with out of the box settings.

2.1 Avoid using `sched_yield`

A good description on the problem of using `sched_yield` on Realtime was written [3]. After discussing the problem of "trying to help the scheduler," the author notes:

> One example can be found in recent changes to the iperf networking benchmark tool, that when the Linux kernel switched to the CFS scheduler, exhibited a 30% drop in performance. Source code inspection showed that it was using `sched_yield` in a loop to check for a condition to be met. After a fix was made, the performance drop disappeared and CPU consumption went down from 100% to saturate a gigabit network to just 9%. [4]

The use of sched_yield is not recommended in Realtime and that serves as a good example how systemtap can be used to identify usage of a certain system call or kernel function and account the usage:

```
#! stap

# pid, process_name and number of calls
# to sched_yield()
global process_list

probe syscall.sched_yield {
  p = pid()
  e = execname()

  if (process_list[p,e])
    process_list[p,e] += 1
  else
    process_list[p,e] = 1
}

probe end {
  foreach ([pid+, name] in process_list)
    printf ("%s[%d] called sched_yield %d times\n",
       name, pid, process_list[pid, name])
}
```

This systemtap script will run until `Ctrl+C` is pressed. Once interrupted, this systemtap script will inform you of the processes that have called sched_yield() and the number of times it was called by each process. For example:

```
[root@lab tmp]# stap sched_yield.stp
ping[2848] called sched_yield 14 times
```

2.2 Bad Priority Assignment

Due to PREEMPT_RT nature, interrupts are threaded and so are several other kernel subsystems. A user application running at the highest available priority, or a priority high enough to be above certain kernel subsystems, if not carefully crafted, can freeze the system. A good piece of advice is to *avoid running applications at the highest available priority*.

A simple example of this case would be the highest priority application running a busy loop, creating statistics that will be sent to disk before the loop ends. That was what probably happened with the gtod_latency test in this email thread [5], where disk IRQ thread was unable to run until gtod_latency finished its busy loop.

The opposite case, where the application under test runs at the lowest priority, or on a priority so low that any

other process could preempt it, is also a disaster if the application is expected to have a high throughput. Observing /proc/<pid>/status can provide valuable information, especially in the nonvoluntary_ctxt_switches field. This field keeps track of the number of times this process has been preempted by other tasks, so a high number in this field would result in high latencies in process because everytime a process gets preempted, its work is interrupted until it gets rescheduled to run again.

This is the easiest case to observe and fix. However, sometimes this problem can be hid when several threads have reasonable priorities except for one with an unacceptable priority. The common way of verifying this is just a matter of using ps:

```
[root ~]# ps -emo pid,tid,policy,pri,rtprio,cmd
...
3826     - -     -     - /usr/lib64/.../firefox-bin
   -  3826 TS   19     - -
   -  3848 TS   19     - -
   -  3849 TS   19     - -
   -  3855 TS   19     - -
   -  3856 TS   19     - -
   -  3857 TS   19     - -
   -  3869 TS   19     - -
   -  8854 TS   19     - -
...
```

What is the best priority to use? That depends on various factors such as what the application does, how it works, what the most important resource for application is , and the like. It should be noted here that the system can be fine-tuned to favor the IRQ or softirq that is most important for the application.

Another important thought to bear in mind is: what will be the scheduler policy of use? Will that be SCHED_FIFO or SCHED_OTHER? Changing scheduler policy and priorities can enhance performance or turn your application into a Denial-of-Service tool.

The tool used to run a given application with the desired priority and under the desired scheduler policy is chrt. This tool can also be used to modify priority and scheduler policy of running processes. It is also possible for the application to set these parameters by itself.

For some of the tests, running the test application at a higher priority and with a better scheduler policy solved the performance issue.

2.3 Resource Allocation

There is a saying in realtime stating that every needed resource should be allocated in advance. Dynamic allocations are prone to resource contention and other latencies which are the worst offenders in realtime.

Whenever an application requests memory allocation, the kernel tries to carry this task as fast as possible. Eventually, these memory pages may not be available and a certain level system reorganization will be required, delaying the requested memory delivery. Sometimes not all requested memory pages are ready to use and when accessed, if not ready, they may trigger a *minor page fault*. In this case, the kernel will need to perform a few steps to solve the situation. This scenario can get more complicated, and the requested memory allocation could trigger a *major page fault*, where the kernel may need to swap memory pages of another application in order to free memory pages for the requesting application. These latencies can be big enough to hurt the realtime constraints of the application.

This case can be even worse because it involves I/O accesses and depending on the system configuration, it may become a nightmare. Imagine a system using a network file system where in order to get the requested memory pages, the system has to flush some buffers through the network. In the context of realtime applications that may lead to unbounded latency spikes and the end of determinism.

Another interesting point is that when a process runs into a page fault it will be frozen, with all its threads, until the kernel handles the page fault.

There are several ways to address this case [6], with the use of mlock() and related functions being the common solution. Unfortunately, the real solutions here require changes in the source code.

Information about the amount of minor and major page faults that already happened to a given application can be obtained using:

- /proc/<pid>/stat – this file has a huge amount of data about the process indicated by pid. There are four fields presenting the number of minor and major page faults faced by this process and the number of page faults faced by its child

```
# if you have problems hooking on exit_mm, use profile_task_exit instead.

probe kernel.function("exit_mm") {
  printf("%s(%d:%d) stats:\n", execname(), pid(), tid())
  printf("\tPeakRSS: %dKB  \tPeakVM: %dKB\n",
    $task->mm->hiwater_rss*4, $task->mm->hiwater_vm*4)
  printf("\tSystem Time: %dms  \tUser time: %dms\n",
    $task->stime, $task->utime)
  printf("\tVoluntary Context Switches: %d  \tInvoluntary: %d\n",
    $task->nvcsw, $task->nivcsw)
  printf("\tMinor Page Faults: %d  \t Major Page Faults: %d\n",
    $task->min_flt, $task->maj_flt)
}
```

Figure 1: Simple script for observing resource usage by processes

processes. The position of this information may change from kernel version to kernel version and it is recommended looking for "contents of the stat file" in the file `filesystems/proc.txt` in your kernel documentation.

- `getrusage` – this function returns resource usage information for a given process. Although not all information fields are available in Linux, page fault information is present.

- `task_struct` – the two methods above gather information from processes' `task_struct`. It is possible to write a simple systemtap script to get this information directly from the source. Later in this session, we will present a systemtap script that gets page faulting information when a given process exits.

Some system calls are known to generate page faults; e.g., `fopen` calls `mmap` to allocate memory which can lead to a page fault. Some are known to present latencies or to have a time resolution that may not be good enough for some applications such as `select` timeout mechanism which uses a `jiffie` based timer. Careful thought and engineering can overcome these issues. A simple approach is to have a separate thread to deal with file operations and other latency prone tasks.

Creating threads on-the-fly can also be a problem if there is a constraint for the time between the event that should trigger thread creation and this new thread handling the event. The same thought is valid for creating new processes through `fork()`.

As already said, there are several ways to get resource usage information, such as: gathering information from `/proc/<pid>/status` in regular time basis; using the `getrusage` infrastructure; or writing a systemtap script that gathers information. One important point to keep in mind when choosing the method is *when* would be the best moment to get the information. Maximum memory usage would be best acquired when the process exits. Time spent in thread creation should be measured during thread creation. The first two methods are harder to synchronize with specific events. Systemtap scripts are more flexible and powerful.

The systemtap script below was used to observe resource usage by processes. When a process finishes, the script prints the maximum amount of real and virtual memory used by the application, CPU time used, voluntary and involuntary context switches, and how many page faults occurred during process execution. The script is simple, and shown in Figure 1.

Two interesting notes about the systemtap script in Figure 1:

- this script gathers information from processes' `task_struct`, and it would be easy to show more information that may be of interest for the user if needed.

- when a process finishes, it calls `exit()` that calls `do_exit()`. The script hooks on `exit_mm()` that is halfway in `do_exit()`—meaning that *after* the script collects data, there still are a few extra steps until the process finishes. The only data that may be different is the system time as the process will spend more time until it really vanishes.

This script was key to observe that processes were spending a long time inside of `do_exit`—in some cases, up to 10 seconds. That helped understanding why performance comparisons carried using the `time` command were so horrible on PREEMPT_RT. The urge to fix this issue was placated by the perception that the process is exiting, it is already out of the run queue and its resources will be freed when needed and when the kernel can do that without disturbing the system.

Resource allocation also comprehends CPU allocation and CPU isolation. For an application with strict real-time constraints, that makes perfect sense reserving one or more CPU in a SMP machine. In some cases it is enough to be sure that thread A and thread B will run in different CPU—that can be easily achieved using the `taskset` command. Sometimes it is necessary to be sure that nothing else will run in a given CPU set, only the application—in this case the best solution is to use the `isolcpus` parameter in the kernel boot command line [7].

When taking in account the fact that PREEMPT-RT has more kernel threads running simultaneously due to threaded IRQ and softirq, it is clear that there are more processes competing for CPU time and that the scheduler works harder to accommodate all tasks. CPU isolation is a valuable tool especially when the application under test does not follow the guidelines discussed in this paper.

2.4 Using libautocork

What to do when a customer application shows 60% performance degradation when running on PREEMPT_RT? And what if after careful system tuning the performance degradation is about 40%?

As the application was network based, the first idea was asking the customer all the information about packet sizes, interval between packets, socket flags in use, and everything else that could give clues to solve the issue. Of course, source code was requested for inspection. Most of the specific questions were not answered and access to source code was denied due to internal security policies.

Several attempts to reproduce the problem in the laboratory, based on the information available, failed miserably. At this point, tests were carried by the customer and our main concern was to not abuse or bother a customer that was as cooperative and helpful as possible.

Several small systemtap scripts were written to get socket information, packet size, protocols in use and everything else. At a certain point Nettaps [8] was created, a set of tapsets and systemtap scripts collecting ideas discussed during the tests and a mix of the most relevant scripts already written. These scripts were sent to the customer and he was able to provide us valuable information on the internals of his application and runtime information.

The `drill.stp` [8] script collects information on the number of times each thread calls `writev`, `poll`, `select`, and `sched_yield`. It also collects statistics on lock contention, per thread and per lock. The last bit of information provided by this script is detailed information on network connections, including buffer size, packet size, average sizes for both buffers and packets, the status on Nagle usage, delayed ACK and similar information. An excerpt from a `drill.stp`'s test run:

```
[root@lab nettaps]# stap -I tapset drill.stp
thread: tid name nrwritev nrpoll nrselect
nrsched_yield
thread: 1934 auditd 6 0 6 0
thread: 1935 auditd 0 0 0 0
thread: 2652 sshd 0 0 4 0
thread: 3300 sshd 0 5 0 0
thread: 3301 sshd 0 0 8 0
thread: 3302 sshd 0 1 37 0
lock: tid futex nrcontentions avg min max
lock: 1934 0x000055555576d508 1 43672 43672 43672
lock: 1935 0x000055555576d534 1 1960 1960 1960
connection: tid saddr sport daddr dport nrbf
avgbfsz minbfsz maxbfsz nrpkts avgpktsz minpktsz
maxpktsz nagle da ka wr
connection: 2652 192.168.0.200 22 192.168.0.100
38915 2 392 176 608 2 392 176 608 0 0 0 2
connection: 3300 192.168.0.200 22 192.168.0.100
36042 1 20 20 20 2 10 0 20 0 7 0 5
connection: 3301 192.168.0.200 22 192.168.0.100
36042 7 297 32 744 8 260 0 744 0 7 0 5
connection: 3302 192.168.0.200 22 192.168.0.100
36042 17 58 48 96 15 57 48 96 0 7 0 5
```

The second script sent to the customer, also part of Nettaps, was `lnlat` [8]. This script gathers information about packet flow, such as time spent by a packet when traveling from the network interface up to the user space application waiting for it, time between a user space application sending a packet and that packet reaching the network, network stack latency, and buffer size statistics. Running `lnlat` produces data like Figure 2.

```
[root@lab nettaps]# stap -I tapset lnlat.stp

latency(ns)        buffer size
  entry  local address  port  remote address port      avg     min   max   avg   min   max
user_in      10.0.0.152 43667     10.0.0.152 20975    38802   38802 38802     0     0     0
user_in      10.0.0.164 20975     10.0.0.164 52965    37717   37717 37717     0     0     0
user_in      10.0.0.164 51375     10.0.1.234  2222    35590   35590 35590     0     0     0
user_in      10.0.0.164 20975     10.0.0.164 52964    51630   51630 51630     0     0     0
user_in      10.0.0.164 20975     10.0.0.164 52995    32706   32706 32706     0     0     0
user_in      10.0.0.152 20975     10.0.0.152 43687    32676   32676 32676     0     0     0
user_in      10.0.0.152 20975     10.0.0.152 43666    50457   50457 50457     0     0     0
user_in      10.0.0.164 37451     10.0.1.234  2222    40151   36293 44010     0     0     0
tcp_out      10.0.0.164 20975     10.0.0.164 52966     1750     818 8150     48     0   232
tcp_out      10.0.0.164 52966     10.0.0.164 20975    12360    6260 26967  7604   624 16384
tcp_out      10.0.0.152 20975     10.0.0.152 43666     1708     928 2488     20     0    40
tcp_out      10.0.0.152 43666     10.0.0.152 20975     5404    5404 5404     40    40    40
tcp_out      10.0.0.152 20975     10.0.0.152 43668     1710     837 7104     48     0   232
tcp_out      10.0.0.152 43668     10.0.0.152 20975    13161    6734 33385  7604   624 16384
tcp_out      10.0.0.152 20975     10.0.0.162 41931   197615    8602 1154511 2585  496  4344
tcp_out      10.0.0.164 52995     10.0.0.164 20975     1078     853 1589      3     0    40
tcp_out      10.0.0.164 20975     10.0.0.164 52995     3493    1042 6633    713     0   840
tcp_out      10.0.0.164 51371     10.0.1.234  2222     1876    1236 3082     20     0    34
tcp_out      10.0.0.152 43687     10.0.0.152 20975     1144     864 1707      3     0    40
tcp_out      10.0.0.152 20975     10.0.0.152 43687     3461     850 6736    713     0   840
tcp_out      10.0.0.164 20975     10.0.0.164 52965     1478     931 2025     20     0    40
```

Figure 2: Output from `stap -I tapset lnlat.stp`

Analyzing information received from the customer, that was not the one in the examples just shown, we were able to understand that:

- customer application was TCP/IP based;
- customer application was using the default `NAGLE` algorithm [9];
- sending millions of small-sized packets;
- sending packets in a burst, without inter-packet interval.

Being the most used transport protocol poses a fantastic challenge for TCP to meet many different needs. Several heuristics were introduced over time as new application use cases and new hardware features appeared and as well kernel architecture optimizations were implemented. In an impressive way, default TCP/IP settings are able to satisfy most users.

As one example of the heuristics in use, TCP delays sending small buffers, trying to coalesce several before generating a network packet. This normally is very effective, but in some cases this heuristic is not the better fit.

Applications that want lower latency for the packets to be sent, especially when working with small packets, will be harmed by this TCP heuristic. There is a knob for applications that want to avoid this algorithm, a socket option called `TCP_NODELAY`. Applications can use it through `setsockopt` sockets API:

```
int one = 1;
setsockopt(descriptor, SOL_TCP, TCP_NODELAY,
   &one, sizeof(one));
```

But for this to be used effectively, applications must avoid doing small, logically related buffer writes as this will make TCP send these multiple buffers as individual packets, and `TCP_NODELAY` can interact with receiver optimization heuristics, such as ACK piggybacking, and result in poor overall performance. In other words, both sender and receiver have to be in tune.

There are several sources of latency for TCP connections [10] and sometimes the solutions applied to a given operating system network stack may not be the best one for another. This becomes clear when working with applications ported from one operating system to another.

It was suggested to the customer using `TCP_NODELAY` to solve, at least partially, the performance drop. But the customer argued that even though the change seemed to

be simple, touching the code was not an option. If we could prove him that a considerable performance gain would be perceived he could try convincing his managers.

Using ideas discussed when trying to correlate data obtained from the Futex Contention systemtap example script [11] and the actual locks in the application, a Glibc stub was written to set `TCP_NODELAY` on the sockets used by a given application. The results were satisfactory but the behavior of `TCP_NODELAY` was not the optimal way of enhancing these connections. Then libautocork [12] was written.

The TCP socket option called `TCP_CORK` is a less known option, present in a similar fashion in several OS kernels, sometimes under a different name. The purpose of this option is to put a cork in the socket, preventing packets from being sent through this socket. When the application decides it is time to send the packet or packets, it just removes the cork.

Applying the cork to a socket is just a matter of setting `TCP_CORK` with a value of 1, as in this excerpt of code:

```
int one = 1;
setsockopt(descriptor, SOL_TCP, TCP_CORK,
   &one, sizeof(one));
```

That tells TCP to wait for the application to remove the cork before sending any packets, just appending the buffers it receives to the socket in-kernel buffers. That allows applications to build a packet in kernel space, something that can be required when using different libraries that provide abstractions for layers.

One example is on the SMB networking protocol code, where headers are sent together with a data payload, and better performance is obtained if the header and payload are bundled in as few packets as possible.

When the logical packet was built in the kernel by the various components of the application, it is time to remove the cork and send the packet. It is important to note that the kernel does not have a way to identify, on behalf of the application, that the packet is ready and must be sent. To remove the cork the application uses:

```
int zero = 0;
setsockopt(descriptor, SOL_TCP, TCP_CORK,
   &zero, sizeof(zero));
```

That makes TCP send the accumulated logical packet right away, without waiting for any further packets from the application, something that it would do in other cases, to fully use the network maximum packet size available.

In order to verify the impact of `TCP_CORK` on the customer application, without access to source code, we decided writing a library to be preloaded and interfere in a few socket related operations. There were other options, such as a systemtap script to touch the system calls related to the functions intercepted by libautocork but the preloaded library offered lower overhead and was simpler to write.

Using libautocork is just a matter of:

```
LD_PRELOAD=libautocork.so ./customer_app
```

Libautocork applies the cork to the sockets and automatically removes it whenever the application waits for an answer to what has been sent. The removal of the cork happens when the application calls `recv`, `recvmsg`, `select` and similar functions.

2.5 Comparing Apples to Apples

System configuration plays an important role in performance test results. In addition, it is important to compare results in the same environment, changing the minimal set of elements possible.

When a little knob is changed, the system may behave in a completely different way during the test. As these knobs vary from `/proc/sys` writable files to device drivers parameters in modules, it may be a hard job to record the exact environment where a test has been conducted. Even the default values for knobs that were not touched may vary from kernel version to kernel version.

Depending on the nature of the application under test, some knobs have no effect on the test. Some knobs may have a visible effect on the results. Determining all the available knobs and their values and eventually recording these values to replay a test or perform another test in the same environment is surely a difficult task.

Some effort has been done to address this need and projects like Tuna [13] and AIT are good examples of such tools. The authors recommend using similar tools to ease the task of comparing results, defining best configurations and even identifying regressions.

2.6 Conclusion

Testing applications under PREEMPT_RT was, most of time, a refreshing and enlightening task. Understanding why a given application presented such a different behavior or performance just by changing the kernel or why some race conditions and bugs were more likely to be triggered under PREEMPT_RT was really insightful. In fact, even kernel subsystems and device drivers had bugs and race conditions uncovered by PREEMPT_RT. The bugs were there all the time, but PREEMPT_RT was more likely to trigger them—this way, several bugs were found and fixed.

On the other hand, seeing all the theories about a given problem falling apart, being unable to reproduce a bug, having to solve a negative performance hit presented by an application without access to information or the source code was sometimes hard to manage. It was even harder to reproduce the exact test environment or the best configuration found without a tool. These moments required creativity and revived the joy of coding and hacking.

The authors hope the reader will benefit from the system tuning and development techniques briefly described in this paper. Hopefully, the readers will even feel inspired by the testing ideas explained through the text and endure the process of creating their own test devices and tools or even automating "bad coding techniques" checking. We hope that ultimately, the reader will find the tools presented here helpful and use them to solve his/her problems.

References

[1] Steven Rostedt and Darren V. Hart, "Internals of the RT Patch," *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, 2007, pp. 161–172.

[2] Real-Time Linux Wiki, `http://rt.wiki.kernel.org`

[3] Arnaldo Carvalho de Melo, *Techniques that can have its behavior changed when the kernel is replaced*, `http://oops.ghostprotocols.net:81/acme/unbehaved.txt`

[4] Ingo Molnar, *Re: Network slowdown due to CFS*, `http://lkml.org/lkml/2007/9/26/132`

[5] `http://www.mail-archive.com/linux-rt-users@vger.kernel.org/msg02364.html`

[6] *HOWTO: Build an RT-application*, `http://rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application`

[7] Kernel boot parameters, `Documentation/kernel-parameters.txt`

[8] Nettaps, `http://oops.ghostprotocols.net:81/acme/nettaps.tar.bz2`

[9] Wikipedia, *Nagle's algorithm*, `http://en.wikipedia.org/wiki/Nagle's_algorithm`

[10] Robert A. Van Valois, Todd L. Montgomery, Steven R. Wright, and Eric Bowden, *Topics in High-Performance Messaging*, `http://www.29west.com/docs/THPM/thpm.html#TCP-LATENCY`

[11] Systemtap War Stories: Futex Contention, `http://sourceware.org/systemtap/wiki/WSFutexContention`

[12] Libautocork, `http://git.kernel.org/?p=linux/kernel/git/acme/libautocork.git;a=blob_plain;f=tcp_nodelay.txt`

[13] Arnaldo Carvalho de Melo, "If I turn this knob...what happens?," *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, 2008.