UDP for Real-Time Linux

Andreas Platschek
Technische Universität Wien
Karlsplatz 13, 1040 Wien, AUSTRIA
a.platschek@wavenet.at

Abstract

This work describes the integration of the hard real-time UDP/IP stack rtludp-0.1.1 into the Realtime Ethernet Device Driver (REDD) of the free real-time Linux distribution RTLinux/GPL. The current implementation of REDD misses a high-level communication protocol as it is present for non-real-time TCP/IP. A high-level communication protocol is essential for high-level applications and middleware (e.g. for using REDD for a publisher/subscriber protocol). This paper describes the design decisions in integrating rtludp-0.1.1 into REDD, analyzes the impact of the layering of the communication protocol on the bandwidth and the latency by means of benchmarks, and points out some ideas for future development.

1 Introduction

This paper illustrates why rtludp-0.1.1 was ported to REDD, describes the most important design decisions and analyzes the impact of a layered communication protocol compared to RAW IP, on the latency by means of simple latency and throughput benchmarks.

In RTLinux/GPL have been several projects with the aim to provide a simple Ethernet based communication interface for distributed applications. One of these projects is Realtime Ethernet Device Drivers (REDD)[1, 2], which provides drivers for the popular Realtek 8139 and the 3Com 3c59x ethernet adapter, as well as a POSIX compliant Application Programming Interface (based on read/write but not the traditional socket API) and low level communication protocols like the MAC and RAW protocol has been included in RTLinux/GPL since version 3.2-pre1. The big handicap of REDD was a missing high-level real-time suitable communication protocol as it is present for non-real-time TCP/IP. A high-level communication protocol is essential for high-level applications and middleware (e.g. for using REDD for a publisher/subscriber protocol like ORTE [3]).

rtludp-0.1.1 was developed as part of the OCERA[4] project but never integrated into mainstream RTLinux/GPL due to technical limitations and driver problems. rtludp-0.1.1 included it's own set of drivers instead of utilizing the existing real-time ethernet framework.
Since a real-time ethernet framework exists - which lacks high-level communication protocols, it was a obvious decision to port the UDP/IP(User Datagram Protocol/Internet Protocol) network stack - implemented in rtludp-0.1.1 - to REDD, so that both projects complement each other. The result is rtl_redd_udp, a hard realtime UDP/IP network stack for RTLinux/GPL. It is already included in rtlinuxgpl-3.2 an can be downloaded at www.rtlinux-gpl.org.
The following section `UDP for Realtime` explains why UDP fits better for realtime applications than TCP, after that, the porting of rtludp-0.1.1's UDP stack to REDD is explained in the section after it, and the impact of layered protocols on the performance is discussed along with a brief description of the benchmarking methodology in `Examples and benchmarks`. Finally a conclusion is built and some ideas for future work are pointed up.

2 UDP for Real-Time

The User Datagram Protocol (UDP) is a connectionless high-level communication protocol. It offers better realtime characteristics than TCP, because in contrast to TCP it employs no transparent retransmission or error correction mechanisms which checks for every packet whether it has arrived in-

cluding proper sequence of packets. Nevertheless it can be guaranteed, that no packets are lost in local networks, provided point-to-point topology is in use. For complex topologies end-to-end checks at the application level, as with all UDP based services, is required.

The key data for real-time communication systems, to allow predictability, are the worst case latency (i.e. the time it takes in the worst case for a packet to be sent), and the jitter (the deviation of the best and the worst case). For some applications these system parameters can be very tight. For example, to control the rollers of a paper machine a worst case latency of approximately 0.1ms to 1ms and a jitter of 1μs are required.
With UDP, this boundary data can be detected by benchmarking very easily, with a round trip test. With TCP it would be much more difficult to predict this basic data due to the unknown number of timeouts and retransmissions. Though statistic approaches have been presented and with appropriate design restrictions distributed systems have been successfully realized based on TCP [5]

3 rtl_redd_udp

The porting effort for the existing rtludp can be separated in two main parts, assessment of existing code and selection for reuse and redesign of the interfaces needed for the specifics of REDD.

3.1 Porting rtludp-0.1.1 to REDD

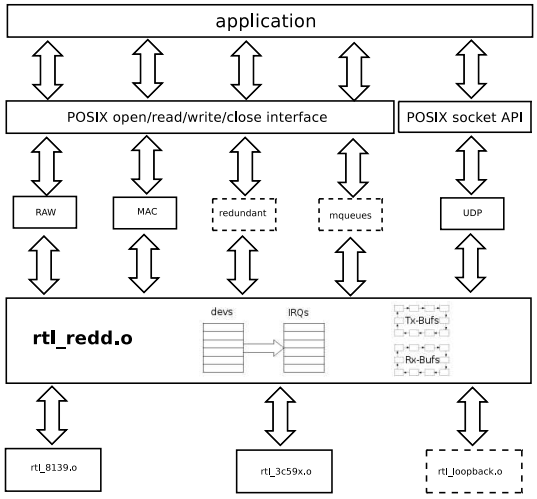

figure 1: *Architecture of REDD*

Before the actual work could be started, the rtludp-0.1.1 code had to be analyzed and it had to be figured out which part of rtludp-0.1.1 should be reused, and which part had to be replaced by the calls to the REDD API.
Basically the porting from rtludp-0.1.1 to rtl_redd_udp meant nothing more than replacing the low-level interface to the drivers of rtludp-0.1.1 by the calls of the REDD API. Of course this does imply evaluating the differences in semantics for the conceptually analogous calls.
So a big part of rtludp-0.1.1 stayed untouched in rtl_redd_udp. This part was mainly the socket interface, which can be found in `udp.c`, and the assembling of the IP- and UDP-Header. As could be expected - pure data manipulation routines could be migrated without much efforts. The resulting architecture of REDD is pictured in figure 1.
Following the UNIX rules of "make it work then fast", after rtl_redd_udp was working, some optimizations have been carried out. One big improvement from rtludp-0.1.1 to rtl_redd_udp was that the pthread_poller which was a periodic task which was polling for messages that should be sent or have been received could be eliminated and replaced by a hardware interrupt service routine, leading to hardware bound latency on packet receive - for packet sending a trigger mechanism has been added - allowing to schedule send operations asynchronously.
With the removal of the message queue included in rtlupd-0.1.1 one Level of buffering was removed. This is not only an architectural and thus maintenance simplification but also a performance gain, as memory related (cache/TLB etc.) operations are the prime source of jitter in RTLinux/GPL.

3.2 The proc Interface

rtl_redd_udp uses the proc interface for two purposes: to read back the current configuration of rtl_redd_udp and to add new entries into the ARP-table. This conceptually follows the rule of only implementing in RT-context what actually is RT sensitive and leaving non-RT configuration operations in non-RT Linux user-space context - with the /proc interface being the simplest sharable data between user-space and kernel-space (which includes RTLinux/GPL threads) this was the option of choice. The corresponding proc-entry can be found in `/proc/redd_udp`, and contains the current IP address of the node, the entries in the ARP-table and a list of the currently opened ports.

rtludp-0.1.1 employs a dynamic ARP-table as used in normal ethernet. This works fine for non-RT applications, but every dynamic part in a real-time system makes it harder to evaluate the systems performance, especially under load conditions, static

resources thus are preferred. It is also a fact that a domain in a real-time system usually consists of a static number of nodes, and so a static ARP-table is used in rtl_redd_udp to resolve the MAC addresses.

While runtime reconfiguration might be necessary, this is considered a non-real-time mode of operation, no temporal guarantees are given during changes in the configuration space of rtl_redd_udp

The ARP-table can either be configured via a configuration file (by default located in `/etc/redd/ARP-table.conf`) or via the proc-write interface. The configuration file is parsed when the module is loaded. The proc-write interface can be used to add entries to the ARP-table, until a socket is opened (i.e. an application (active rt-thread has been loaded).

4 Examples and Benchmarks

During development, some simple examples have been written. If you download rtlinux-3.2 you can find them in `rtlinux-3.2/drivers/redd/examples`. These examples can basically be used as templates for applications, among the examples there are some simple tests to measure the round-trip time and the bandwidth. At the current state of development, the round-trip test works fine, but there are some troubles with the bandwidth, test, so in the following only the round-trip test is discussed.

The output of this test is compared to the output of the raw protocol included in REDD, to allow direct comparison of the impact that a layered protocol could have. (figure 2) in the following. The results shown in this paper were carried out with two 100Mbit rtl8139 ethernet adapters, connected via a cross-over cable. Systems used were COTS PCs, NN 1.6GHz AMD Athlon systems with 256MB RAM and running from NN IDE disks.
It should be noted that while results are hardware dependent, the relation between UDP and RAW connection is expected not to change with different hardware setups.

The round-trip test sends a packet from one node to another node, which sends the packet back unmodified except for exchange of the header information. The time this procedure takes is measured representing the lower bounds of achievable communication time for a given packet size. The measurement is carried out several times and the best-, worst- and average- case are evaluated. Then the packet size is increased and the measurements are performed with the new packet size. Tests showed that results are reasonably stable with 100 packets. The parameters (MAX_SIZE, MIN_SIZE, STEP_SIZE and NUM_MEASUREMENTS) for the round-trip test can be chosen by the user. The result of the round-trip test for rtl_redd_udp is shown in figure3 with best-, worst- and average-case printed against the packet size - this test run is with 100 measurements.

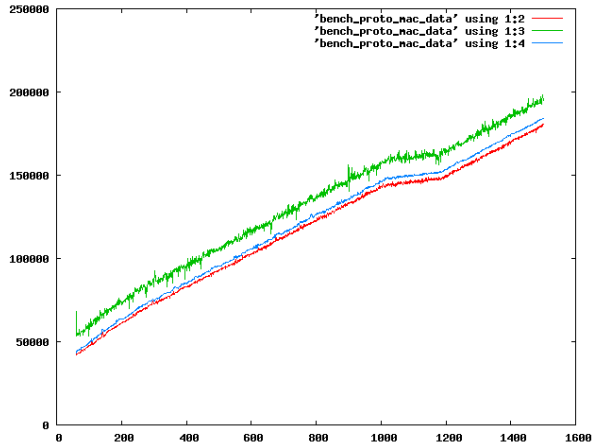

figure 2: *Round-Trip Time RAW protocol*

If we compare the Round-Trip Time of the raw protocol in figure 2 to the Round-Trip time of rtl_redd_udp in figure 3, we can see that the worst case time stays roughly the same (starting with about 50μs at a packet size of 50Byte up to about 200μs at a packet size of 1500Byte), but minimum and average values are a little bit higher in rtl_redd_udp. This is due to the increased code size for processing packets resulting in higher cache related delays and in the overhead due to greater function call depth, though the overall impact is negligible with respect to determinism of communication properties.

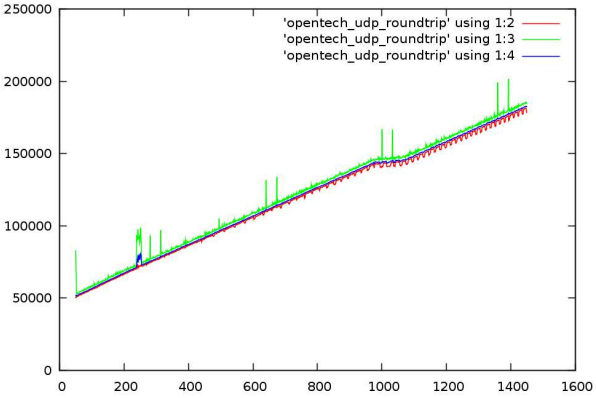

figure 3: *Round-Trip Time UDP protocol*

The bandwidth test does almost the same as the round trip test, but instead of sending one packet, a number of packets is sent, and the time, until

all the packets are received is measured. While the round trip test is dominated by the ethernet interfaces themselves and the CPU side processing can be considered uncritical, the bandwidth tests will be influenced by the CPU and memory subsystem performance especially for large packet sizes.

These tests currently have not been run on sufficiently different platforms to draw significant conclusions regarding the performance impact of the individual hardware components.

5 Conclusion and Future Work

At this point of development, rtl_redd_udp supports a basic function set to use the standard socket API in real-time applications. This makes the use of the communication via ethernet in real-time context a lot easier, and the application programmer is able to use the POSIX socket API as he is used to in non real-time context.
At the same time initial benchmarks show that the impact on determinism is negligible justifying the use of a high-level protocol in real-time context.

For the future it will be necessary to implement multi NIC support, find the reason for the spikes in the worst case and do some optimizations. There are also some restrictions in rtl_redd_udp that should be eliminated. One example is the number of ports used at a time. During test phase this number has been increased up to 40 without any problems. However, to keep the system deterministic, we will follow the concept of static resources, whenever this is possible.

Migration to RTLinux/GPL-4.0 (based on the XtratuM nanokernel) is one of the near future challenges for rtl_redd_udp.

References

[1] F. Bruckner, RTLinux Ethernet Device Drivers, *Proceedings of the Eighth Real-Time Linux Workshop*, pages125-127, 2006

[2] Sourceforge, *REDD: RTLinux Ethernet Device Drivers*, 2006, `http://redd.sourceforge.net/`

[3] Jan Krakora, Pavel Pisa, Frantisek Vacek, Zdenek Sebek, Petr Smolik and Zdenek Hanzalek, *Deliverable D7.4 Communication components V2*, OCERA Consortium, February 2004, `http://www.ocera.org`

[4] , A. Crespo and I. Ripoll, *OCERA White Paper*, OCERA Consortium, April 2003, `http://www.ocera.org`

[5] , G. Alt, W. Lages, *Networked Robot Control with Delay Compensation* Federal University of Rio Grande do Sul, Institute of Informatics, Nov 2003