Hard Real-Time Networking on FireWire

Yuchen Zhang, Bojan Orlic, Peter Visser, Jan Broenink
University of Twente
Enschede, the Netherlands
Yuchen623@gmail.com
B.Orlic@ewi.utwente.nl
P.M.Visser@ewi.utwente.nl
J.F.Broenink@ewi.utwente.nl

Abstract

This paper investigates the possibility of using standard, low-cost, widely used FireWire as a new generation fieldbus medium for real-time distributed control applications. A real-time software subsystem, RT-FireWire was designed that can, in combination with Linux-based real-time operating system, provide hard real-time communication over FireWire. In addition, a high-level module that can emulate Ethernet over RT-FireWire was implemented. This additional module enables existing IP-based real-time communication frameworks to work on top of FireWire. The real-time behavior of RT-FireWire was demonstrated with a simple control setup. Furthermore, an outlook of the future development on RT-FireWire is given.

1 Introduction

1.1 Hard Real-Time Network (fieldbus) in Distributed Control

Computer-controlled systems are more and more often implemented on top of distributed hardware architectures. Compared with a centralized architecture, the distributed architecture can achieve higher computation throughput, enhanced capabilities for fault tolerance and easier maintenance per part of the system. On all distributed control architectures, a common critical issue is the hard real-time communication infrastructure, which is normally refered to as a fieldbus.

Typical distributed control systems come in all topologies and sizes, with widely varying workload imposed by one or more control loops repeated with frequencies typically falling in the range between 100 Hz and 5 kHz. There is no single perfect fieldbus that fits in all types of control applications. A fieldbus will be a suitable choice in the context of a given application if and only if real-time guarantees can be offered to meet the temporal characteristics of message transmissions required by the given application.

Every fieldbus has certain hard limits of its real-time capabilities imposed by characteristics of the network protocol and the hardware part of its implementation. If the protocol/hardware component of a network cannot meet the requirements of the application at the hand, another type of fieldbus should be chosen. However in practice, often the software layer of a network plays a key role in its determinism and applicability in hard real-time systems. When such a software layer is implemented improperly, the promising characteristics of the protocol / hardware part will not be exploited to the maximum extent. Often, software layers are implemented for widely used, but essentially non real-time operating systems, e.g. GNU/Linux. Those operating systems are optimized for maximizing system throughput and improving average performance. Software layers implemented on top of such operating systems will pay the price of being less deterministic than they ought to be.

In this research an attempt was made to implement a hard real-time network based on a standard, widely used and relatively cheap hardware. Ethernet, FireWire and USB are the most widely used communication interface equipped by standard computers of today. FireWire and Ethernet are peer-to-peer networks and thus more suitable for distributed control systems. Several implementations of the real-time software layer on top of Ethernet al-

ready exist[1]. FireWire is on the other hand less researched as a real-time network. Here, RT-FireWire, a real-time software layer above FireWire hardware, currently running on Xenomai[5], is introduced.

The main objective of this paper is to present the design of RT-FireWire and its extensability. Section 2 deals with a brief introduction of FireWire and the inadequacy of using Linux FireWire subsystem[3] for real-time applications. Section 3 explains the most important aspects of the RT-FireWire design, plus the benchmark that illustrates RT-FireWire's usability for real-time applications. Section 4 introduces the implementation of real-time IP network over FireWire. Section 5 briefly discusses a control system study case and analyses the obtained results. At the end of this paper, summary and future work are given.

2 FireWire and Its Subsystem in Linux

2.1 Brief Introduction of FireWire

FireWire, also known as IEEE1394[2] is a high-performance serial bus for connecting heterogeneous devices. Though firstly targeted for consumer-electronic applications, such as high-speed video transmission, FireWire also presents very valuable characteristics for communication in industrial applications, e.g. distributed control system.

The bus topology of FireWire is tree-like, i.e. non-cyclic network with branch and leaf nodes. The physical medium supports data transmission up to 400 Mbps in the 1394a specification. In the 1394b specification, the speed even rises to 3.2 Gbps. Two transfer modes are supported on FireWire: asynchronous and isochronous. As illustrated in Figure 1, a mix of isochronous and asynchronous transactions is performed by sharing the overall bus bandwidth, whose allocation is based on 125 μs intervals, the so-called FireWire cycles.

An isochronous transaction targets one or more nodes by being associated with a multicasting channel number. There can be maximally 64 channels in total. Once the bus bandwidth has been allocated for an isochronous transaction, the associated channel can receive a guaranteed time-slice during every 125 μs cycle. Up to 80% (100 μs) of every bus cycle can be allocated to isochronous channels. Because this transaction type does not re-transmit lost or corrupted packets, but delivers data at constant rate, it is well suited for the time-triggered state message transmission in distributed control systems.

In the asynchronous transaction phase, the whole network on FireWire appears as a large 64-bits mapped bus address space, with each node occupying a 48-bits mapped space. The higher 16 bits are used to identify nodes[1]. An asynchronous transaction is split into 2 sub-transactions: a request to access a piece of address on another node and response. Coordination between request and response is ascertained by the Transaction Layer protocol. Since guaranteed data delivery is provided through acknowledgment, asynchronous transaction is targeted for non-error-tolerant applications, like command and control message transmission in distributed control system.

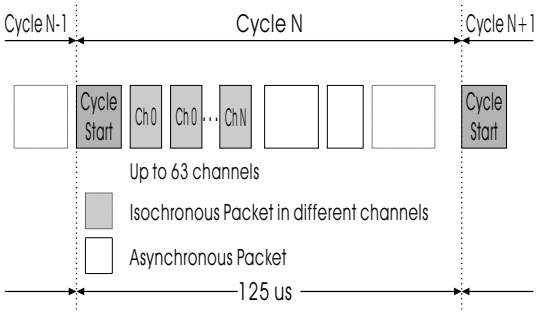

FIGURE 1: *FireWire Cycle*

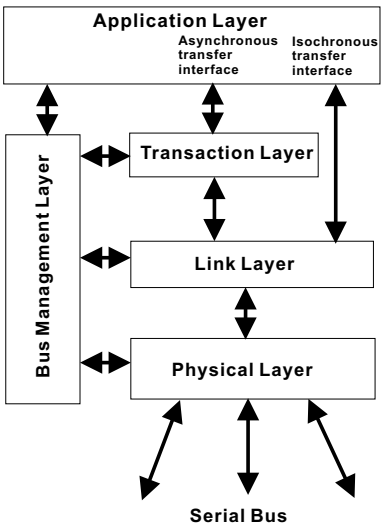

FIGURE 2: *FireWire protocol layers*

As shown in Figure 2, the FireWire protocol defines several layers with specific roles. The Physical Layer deals with transmission medium issues, cabling and signal levels. The Link Layer performs the CRC checks, determines the type of transaction and forwards the received packet to the next layer. The Transaction Layer provides support for asynchronous

[1]Here, we only talk about the peer-to-peer asynchronous transaction. In the 1394a supplement, a multicasting packet in asynchronous transaction is also defined.

transactions and relies on the services of Link Layer to achieve this. Isochronous transfers are directly issued to Link Layer. The bus management layer takes care of some bus level functions: isochronous bandwidth allocation, power management and cycle master management. Cycle Master is the role taken by a node that initiates start of the bus cycle by sending a cycle start message.

2.2 Performance Benchmark on Linux FireWire Subsystem

The first step was to explore the usability of the Linux FireWire subsystem in real-time context. A benchmark was used to evaluate the system's performance in 4 cases: both isochronous and asynchronous transactions were tested under light and heavy system load.

In each case, two nodes were involved (as shown in Figure 3): one was the requesting node that was actively sending the data; another was the target node that was passively receiving the data. In case of asynchronous transaction, the target node generates a response packet and sends it to the requesting node. In asynchronous case, the transaction latency was measured. In isochronous transaction the data sending rate was 1 kHz. and the drift of the subsequent interval of receipts from the expected 1ms was measured. For each case, 100,000 data samples were collected for analyzing. During the experiment, the data payload was always 56bytes.

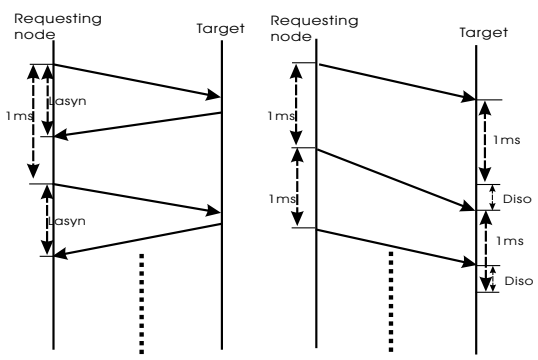

FIGURE 3: *Experiments on FireWire*

To conduct the experiment under heavy loaded condition, extra processing load needs to be imposed explicitly. The combination of three ways to impose system load was used in this experiment.

- Creating a flood of interrupts from external world by using a third node to send a lot of random data to the nodes in experiment via Ethernet.
- Creating a flood of interrupts from hardware disk I/O by reading the whole hard disk.
- Creating a flood of system calls via Linux command line, which makes a lot of kernel-user context switch.

The test bench setup employs two PC104 stacks connected via FireWire. The used PC/104 boards has VIA Eden 600 MHz processor, 256 Mb Memory, 32 Mb flash disk. The used FireWire Adapter is PC/104 board with VIA VT6370L Link & Physical LayerTransaction Layer chip, supporting 400 Mb/s transferring speed at maximum. On the software side, Linux kernel version 2.6.12 is used.

2.3 Experiment Results

The result is presented by using cumulative percentage curves. At any point on the cumulative percentage curve, the cumulative percentage value (y-value) is the percentage of measurements that had a latency less than or equal to the latency value (x-value). The latency at which the cumulative percentage curve reaches 100 percent represents the worst-case latency measured. For real-time transaction latency, the ideal cumulative percentage curve is one that is steep with a minimal decrease in slope as the curve approaches 100 percent. The cumulative percentage at a certain latency value can be interpreted as the probability of the transaction being able to meet real-time constraints when its deadline is assumed to be equal to that latency value. For example, let's imagine a networked control system (with dominant communication delay) running with 10kHz sampling frequency. If the control system is designed in such a way that control delay is exactly the same as sampling period, then cumulative percentage latency of 97% would mean that in 97% of the cases control action can be performed in time.

As can be seen from Figures 4 and 5, when the system is not loaded, the experiment results on either asynchronous or isochronous transactions already indicate a relatively big difference in latency values (asynchronous case) or receiving rate drift (isochronous case) in the critical range of cumulative percentage (e.g. between 97% and the worst case (100%) performance). With the load added, the performance is clearly worsened. Moreover when the system is heavily loaded, the curve is much less steep than in the case system is not heavily loaded. As it can be concluded from previous discussion, this indicates increased non-determinism and poor real-time properties.

For real-time application, it is the worst case (or almost worst case, like 99.999% threshold) that drives the choice for underlying system. And for typical real-time control application, e.g. high-speed

motion control applications, the measured worst case performance can not satisfy the requirements. Therefore, the conclusion can be reached: Linux FireWire subsystem can not be used as underlying networking platform for real-time control application. Hence, there is a need to develop a special FireWire subsystem for use in real-time control application.

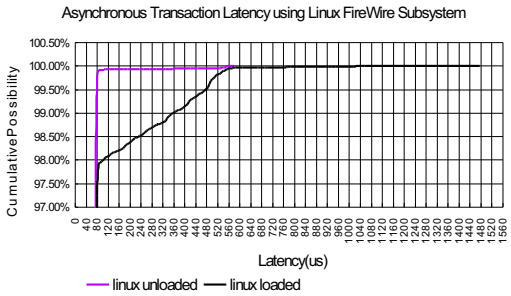

FIGURE 4: *Asynchronous transaction latency of Linux FireWire subsystem*

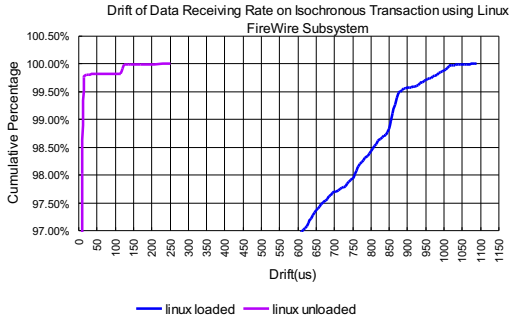

FIGURE 5: *Isochronous transaction drift of Linux FireWire subsystem*

3 RT-FireWire

In this section, the implementation of RT-FireWire is presented, including the system overview, the architecture and the core components. Secondly, the results from repeating the same benchmark (Figure 3) on RT-FireWire is given, as a illustration about the significantly improved real-time performance of RT-FireWire as compared with the original Linux FireWire subsystem.

3.1 System Overview

RT-FireWire has the Linux FireWire subsystem as the starting point. Therefore, it exhibits all the function blocks of the original Linux FireWire subsystem, as shown in Figure 6. Besides, two new function blocks are added: real-time memory management and RTcap. RTcap stands for Real-Time (Packet) Capturing, which is designed to capture all incoming and outgoing packets in order to facilitate network behavior analysis. The core of RT-FireWire is implemented over the RTDM (Real-Time Driver Model) skin[9] of Xenomai.

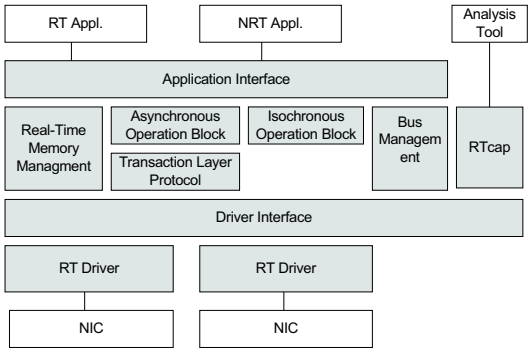

FIGURE 6: *RT-FireWire kernel*

3.2 Task Composition

The architecture of RT-FireWire is strictly divided into several layers: Hareware Operation layer, Protocol Processing layer and Application layer. Each of these layers corresponds to one layer in the network protocol of FireWire: respectively Data-Link Layer, Transaction Layer and Application layer. Each layer is composed of one or more real-time tasks, which are schedulable objects in the system scheduler. All these tasks are servers that handle asynchronous events. The task composition in RT-FireWire's layered architecture is shown in Figure 7.

Interrupt Broker is the task that belongs to Hardware Operation Layer. It handles various FireWire bus events: receipts of asynchronous transaction request and associated acknowledgment, receipt of asynchronous transaction response and associated acknowledgment and up to 64 events for the data receipts via isochronous channels, etc.

The Protocol Processing Layer consists of several tasks/brokers that perform essentially the same service (related to services of Transaction Layer protocol) but for packets of different priorities assigned according to the temporal requirement of applications. The mechanism of prioritizing is addressed in Section 3.3.

The Application Layer has two types of brokers designated to handling packets of the two transfer mode. Those brokers allow applications to customize callback functions that can handle packets in the ap-

plication specific ways.

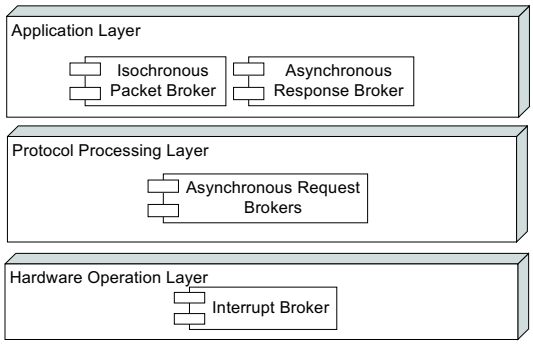

FIGURE 7: *Layered task structure of RT-FireWire*

3.3 Real-Time Transactions

All asynchronous FireWire packets are prioritized. Priority consists of the last 4 bits (16 priority levels) in the first quadlet of asynchronous packet. Originally, in the specification of FireWire protocol[2] these 4 bits are reserved for backplane environment. However, since RT-FireWire only aims to be used in cable environment, those bits are free to be used for alternative purpose. In the current implementation of RT-FireWire, the priority 0 (highest) is assigned to packets for bus internal management service; the priority 15 (lowest) is assigned to non real-time applications; the priorities in the between are assgined to real-time applications. In the Protocol Processing Layer, three brokers are employed to handle the packets of the three ranges of priority, namely the Bus Internal Service Broker, the Real-Time Broker and the Non Real-Time Broker, as shown in Figure 8. The Real-Time Broker has the priority sorting of its packet queue. The Non Real-Time Broker stays in Linux domain as a signal handler, therefore RT-Firewire sends the corresponding signal to Linux upon the arrival of a packet with lowest priority.

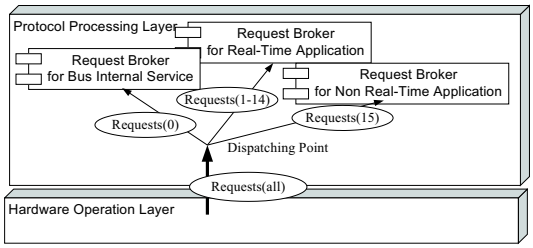

FIGURE 8: *Brokers in Protocol Processing Layer*

Stamping the priority levels into each packet also enables the priority sorting in packet sending queues, which allows, at least on the software layer, that transactions of higher priority can preempt the transactions of lower priority.

Compared to the NIC(Network Interface Card) driver in Linux FireWire subsystem, the driver in RT-FireWire is extended with the capability to provide accurate timestamp services for both incoming and outgoing packets. For incoming packets, the reception time is stamped onto the packet object in the beginning of interrupt handler (by assigning the value of current time to the reception time element in the packet structure). For outgoing packets, the driver also provides the functionality to store the value of current time into a certain part of the sent data (as transmitting time) just before stuffing the packet to hardware (via DMA).

3.4 Real-Time Memory Management

The implementation of Real-Time Memory Management relies on preallocated memory management pools, which is inspired by the design of socket buffer management in [1]. The task receiving a packet from another task must provide a memory buffer from its own memory pool for packet compensation. In this way, the tasks that have own memory pools are independent of load conditions in other parts of the system - no task can cause unexpected memory starvation of other tasks in the system.

In Figure 9, the distribution of memory pools inside RT-FireWire is illustrated, which corresponds to both layered structure of FireWire and the task division among Broker objects.

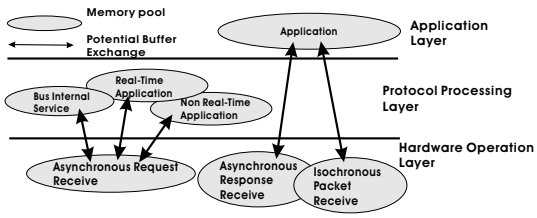

FIGURE 9: *Layered distribution of memory pools inside RT-FireWire internals*

A real-time packet buffer consists of a management header and the body of the data buffer. The header further contains the generic part and the protocol-specific part. The memory management module operates only on the generic part of the packets header, while the FireWire-specific modules operate only on the protocol-specific part. This way, the operations on both sides are transparent to each other, therefore the management module can always keep the global control and monitoring of the memory usage in the whole system regardless of the FireWire-specific operations or the buffer exchanging between FireWire protocol layer and modules at the application layer.

3.5 Real-Time Packet Capturing

This service consists of two parts: packet capturing module in the kernel side and the analysis tool in the user side. The kernel-side module captures both incoming and outgoing packets in the "Captured Packet Queue". Each captured packet is waiting either for the processing by some applications or the processing by analysis tool. Memory leaking is prevented by capturing module providing, from its memory pool, a compensation packet for every captured packet. Application relies on the API of Real-Time Memory Management module for deallocating packets, for which the actual implementation in Real-Time Memory Management module will not deallocate the packet but it will instead return the compensation packet to the application-owned memory pool. See Figure 10 for illustration of the whole procedure.

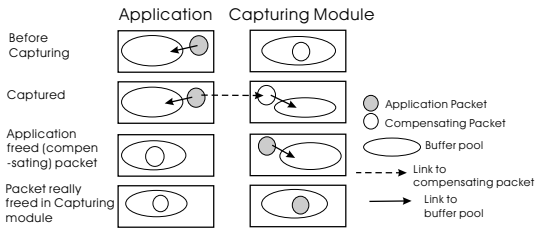

FIGURE 10: *Packet Capturing Procedure*

3.6 Comparison with Linux FireWire Subsystem

Same experiment as the one on Linux FireWire Subsystem was repeated on RT-FireWire, except that the operating system used is changed to patched Linux 2.6.12 plus SVN version of Xenomai (at the time of writing). The results are shown in Figures 11 and 12, together with the results of experiments on Linux FireWire Subsystem in order to make them directly comparable.

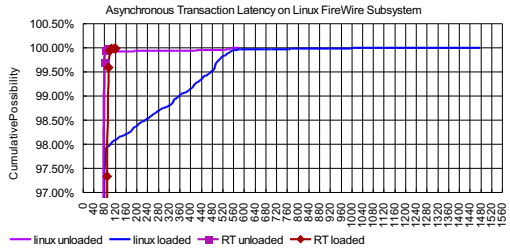

FIGURE 11: *Comparison of RT-FireWire and Linux FireWire subsystem (Asynchronous transaction)*

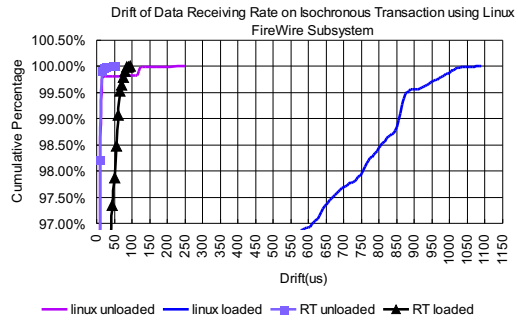

FIGURE 12: *Comparison of RT-FireWire and Linux FireWire subsystem (isochronous transaction)*

As shown in the figures, both the asynchronous latency and the isochronous drift on RT-FireWire have quite steep curves compared to Linux FireWire Subsystem. That means RT-FireWire gives much more deterministic behaviour, which is especially crucial when time critical communication for real-time applications is needed.

4 IP over RT-FireWire

This section explains the implementation of deploying real-time IP network on top of RT-FireWire, and the performance comparison with real Ethernet.

4.1 Ethernet Emulation

The IP over RT-FireWire is enabled via Ethernet emulation, a module in the application layer. It is based on the "IP over 1394" specification[6], which standardizes the way to transfer IP packets via FireWire. The Linux FireWire subsystem contains a highlevel module Eth1394 implemented according to this specification. The Ethernet emulation on RT-FireWire was built by taking the Eth1394 in Linux as the starting point. In addition to some modifications and extensions that were needed to make the Ethernet emulation real-time and compatible with RT-FireWire, the mechanism of address resolution between the IP address and the network address (per FireWire node) is somewhat different than the one proposed in "IP over 1394" specification. One can refer to [10] for relative information.

RTnet[4] is another open source project that provides a customizable and extensible framework for hard real-time communication over Ethernet. RTnet provides its real-time services via real-time variants of POSIX-conforming socket interface. Implementation of Ethernet emulation layer over RT-FireWire has enabled the usage of RTnet on top

of the FireWire medium, which introduces an alternative to Ethernet for real-time IP networking, as shown in Figure 13.

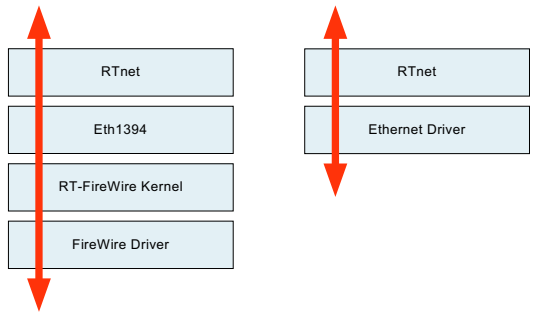

FIGURE 13: *RTnet on top of RT-FireWire*

4.2 Comparison with Ethernet

A test bench was built between two FireWire nodes (equipped with the same hardware and software as described in previous section, plus RTnet 0.9) to measure the roundtrip latency of transactions on the IP layer over RT-FireWire. The same experiments also were repeated on real Ethernet interfaces of these two nodes. The results of both, which were measured when system was heavly loaded, are plotted together in Figure 14.

The results of the test bench shows that RTnet over RT-FireWire gives performance comparable with RTnet over Ethernet. The former shows larger latency and larger latency variation (jitter), i.e. the difference between 97% threshold latency and worst case latency. This is due to the relatively more complex software stack: the packet path through the whole RT-FireWire includes more task handover, context switches, which inevitably cause more lantency to the whole data path. On the other hand, the former has a less leaning slope. This is due to the higher data transfer rate on FireWire: the FireWire devices in experiment can transfer data at 400Mb/s, while the Ethernet devcie in experiment can only transfer at 100Mb/s.

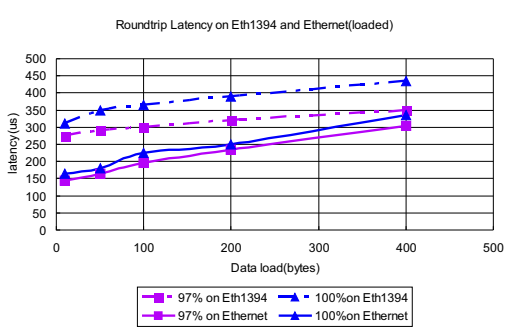

FIGURE 14: *RTnet on Ethernet vs. RTnet on RT-FireWire*

5 Study Case

RT-FireWire was tested with a real, albeit rather simplistic control setup. The plant consisting of DC motor, belt pulley and flywheel was controlled by simple PID controller. The experiment was performed for both centralized and distributed control system. In distributed configuration one PC/104 stack is used as an I/O node that performs sensor (encoder) acquisition and actuator (PWM) signal update. The other node executes the code of controller. In every control loop, the data packets travel both directions from I/O node to computation node and back. The configuration is illustrated in Figure 15.

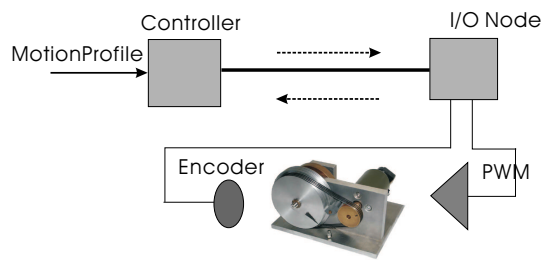

FIGURE 15: *Distributed control of Linix plant (2-Way configuration)*

With 1 kHz sampling frequency, the distributed system works well without missing any deadlines, which means the distributed controller gives exactly the same behavior as the centralized controller. As shown in Figure 16, the plotted curves (of PWM, encoder and profile of motion reference) from distributed control exactly laps over the curves from centralized control.

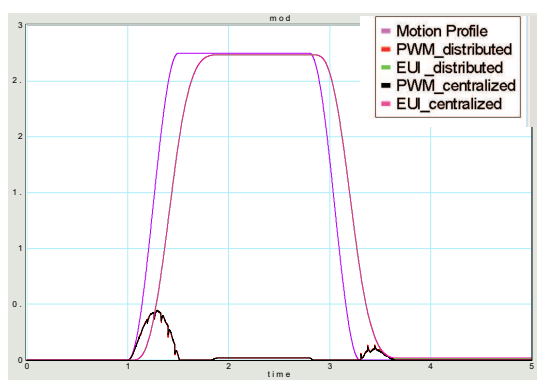

FIGURE 16: *Comparison between centralized control and distributed control (1kHz)*

With 5 kHz sampling frequency, some deadlines are missed. But the measured output of the plant shows almost no difference in the observed behavior compared to the centralized configuration, as shown

in Figure 17. Note, however that this is most likely the case due to the relatively slow dynamics of the plant, whose behavior is almost not influenced by the increase of sampling frequency from 1kHz to 5kHz.

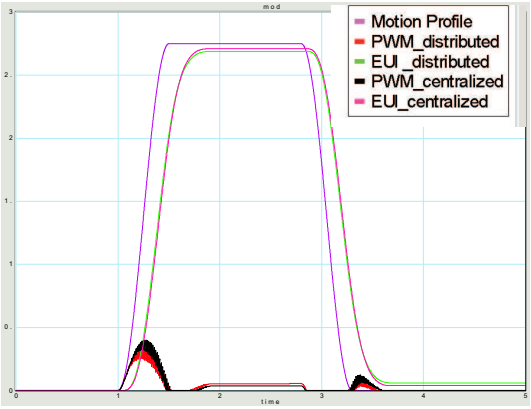

FIGURE 17: *Comparison between centralized control and distributed control (5kHz)*

6 Summary and Outlook

This paper introduces the real-time software subsystem on FireWire (RT-FireWire), which provides deterministic communication over FireWire. The results from the performance benchmarking show that, the transaction latency on RT-FireWire can be limited to a certain range usable for distributed control application, whether the system is under heavy load or not. Ethernet emulation over FireWire has been fully implemented on RT-FireWire as a module in the application layer. Via Ethernet emulation, RT-FireWire can be connected to another real-time software framework, RTnet. Therefore, FireWire can be used as a new medium alternative to Ethernet for real-time IP networking. The performance benchmarking on Ethernet emulation and Ethernet shows that the performance of both is comprable. A study case was carried out to see RT-FireWire's suitability for a "real-life" controlling task. The result illustrates that, RT-FireWire has fast and deterministic behavior that makes FireWire fully usable as a fieldbus for distributed controlling tasks.

Future work will be focused on the development of new modules in the application layer. One branch is to develop a raw interface on RT-FireWire. So via this interface, operation can be directly applied on FireWire layer, e.g. issuing transaction, allocating bus address space or isochronous channels. The current raw1394 module in Linux already implements the similar functions, but of course in a non real-time manner. However it can be a reasonable starting point. Furthermore, the potential of layering other middleware frameworks over RT-FireWire will be addressed. At the time of writing, two options are in the list: one is CANopen, which has been developed (originally for communication over CAN) as an application protocol and device model for the automation domain[7]; another is 1394AP (1394 Automation Protocol), which is an emerging standard to build communication systems for factory automation and motion control via the application layer of IEEE1394[8].

For more internal information of RT-FireWire, one can refer to [10]. Based on the work presented in this paper, RT-FireWire has been converted to an Open Source project, registered at *www.berlios.de*. It can be directly visited via **rtfirewire.berlios.de**

References

[1] J. Kiszka, B. Wagner, Y. Zhang, J. Broenink, 2005, RTnet-A Flexible Hard Real-Time Networking Framework, in *10th IEEE International Conference on Emerging Technologies and Factory Automation*, Catania,Italy.

[2] IEEE1394, 1995, *IEEE standard for a high performance serial bus Std 1394-1995 and amendments*

[3] Linux FireWire Subsystem project homepage *www.linux1394.org*

[4] RTnet project homepage *www.rtnet.org*

[5] Xenomai project homepage *www.xenomai.org*

[6] P. Johansson, 1999, IPv4 over IEEE1394 *RFC2734*.

[7] CiA, 2002, *CANopen, Application Layer and Communication Profile*, CAN in Automation.

[8] K. Frommhagen, P. Nauber, U. Schelinski, M. Scholles, 2005, 1394AP:A Protocol for Deterministic Industrial Communication via IEEE1394, in *10th IEEE International Conference on Emerging Technologies and Factory Automation*, Catania, Italy.

[9] J. Kiszka, 2005, The Real-Time Driver Model and First Applications, in *7th Real-Time Linux Workshop*, Lille, France.

[10] Y. Zhang, 2005, Real-Time Network for Distributed Control, MSc thesis 031CE2005, University of Twente, Enschede, the Netherlands, *www.ce.utwente.nl/rtweb/publications/Msc2005/pdf-files/031CE2005_Zhang.pdf*.