

RTnet – A Flexible Hard Real-Time Networking Framework

Jan Kiszka, Bernardo Wagner
Institute for Systems Engineering
Real-Time Systems Group
University of Hannover, Germany
{kiszka, wagner}@rts.uni-hannover.de

Yuchen Zhang, Jan Broenink
Control Engineering Group
Department of Electrical Engineering
University of Twente, the Netherlands
y.zhang-4@student.utwente.nl,
j.f.broenink@utwente.nl

Abstract

In this paper, the Open Source project RTnet is presented. RTnet provides a customisable and extensible framework for hard real-time communication over Ethernet and other transport media. The paper describes architecture, core components, and protocols of RTnet. FireWire is introduced as a powerful alternative to Ethernet, and its integration into RTnet is presented. Moreover, an overview of available and future application protocols for this networking framework is given.

1 Introduction

Real-time Ethernet has grown to one of the core topics in current industrial automation research and application. A significant number of vendor-driven solutions have shown up on the market during the last years, claiming to replace traditional fieldbuses. The overview of available solutions on [18] currently lists 16 soft and hard real-time Ethernet variants. Most of them either require special hardware extensions to nodes or infrastructure components, or they provide only soft real-time guarantees. Academia approaches are typically designed to demonstrate specific concepts and lack common OS or hardware support. A broad overview of soft and hard real-time protocol research is given in [7]. Some recent approaches are for example FTT-Ethernet [16], RT-EP [12], or the combination of switches and traffic shapers [11].

All these approaches come with various transport and application protocols as well as programming interfaces, which are generally not compatible with each other. Additionally, there are other transport media beyond Ethernet 100Base-T approaching the real-time domain: Gigabit Ethernet, wireless media as IEEE 802.11 or Bluetooth, and also promising trends like using FireWire for time-critical control and measuring tasks. While this diversity of solutions can stimulate competition, it also interferes with the portability and extensibility of applications both in research and industrial scenarios. Furthermore, the question arises which solutions can guarantee long-term availability, especially when taking their spe-

cific hardware dependencies into account.

With the goal to provide a widely hardware-independent and flexible real-time communication platform, the RTnet project has been re-founded in 2001 at the University of Hannover, based on ideas and source code of a previous effort to provide deterministic networking [10]. RTnet is a purely software-based framework for exchanging arbitrary data under hard real-time constraints. The available implementation is founded on Linux with the hard real-time extension RTAI [17].

The design of the RTnet stack as depicted in Figure 1 was inspired by the modularised structure of the Linux network subsystem. It aims at scalability and extensibility in order to comply with the different requirements of application as well as research scenarios. RTnet's software approach addresses both the independence of specific hardware for supporting hard real-time communication and the possibility to use such hardware nevertheless when it is available. Furthermore, it enables the integration of various other communication media beyond Ethernet.

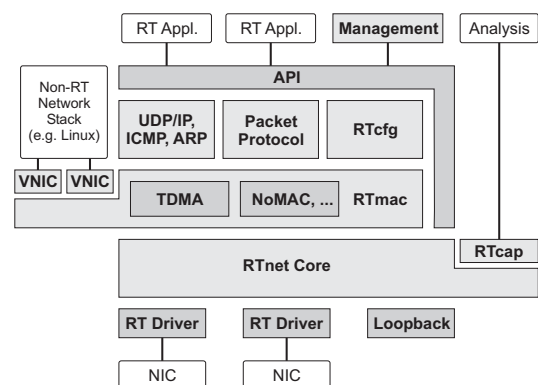


Figure 1. RTnet Stack

This paper presents the architecture of RTnet and the realisation of its central components. Section 2 describes the RTnet base services consisting of the stack core, the driver interface, available transport protocols like the real-time UDP/IP implementation, the programming interfaces provided to management tools and real-time applications, and the packet capturing extension RTcap. The deterministic media access control framework RTmac, including

its tunnelling network devices for time-uncritical traffic (VNIC), is introduced in Section 3. That section will furthermore present RTnet's default access control discipline for Ethernet, TDMA, in details. Finally, Section 4 closes the stack overview by addressing the real-time configuration service RTcfg. So far, the implementation of RTnet has been focused on Ethernet. Section 5 presents the concepts and recent advances to add real-time IEEE 1394 (FireWire) support to the framework. The section also points out the advantages of that media type and the possible applications in the automation domain. Furthermore, available and future application protocols and full-featured middlewares working over RTnet are described in Section 6.

2 Base Services

RTnet contains a set of central services which are required for most scenario. In the following, these service will be introduced.

2.1 Packet Management

One of the crucial parts of RTnet deal with the management of packets which contain the incoming and outgoing data. Packets that ought to be transmitted are passed through the stack in the context of the sending task, i.e. a real-time application or an internal RTnet service. In contrast, incoming packets are first passed from the network controller driver to a so called stack manager. This real-time task demultiplexes the packet according to their protocol types by passing them to the respective handlers. The priority of the stack manager has to be above all applications using RTnet services in order to avoid priority inversions. This concept is similar to bottom-half interrupt handling as it can be found in most operating systems.

The stack and the drivers use a unified data structure called `rt_skb` (derived from the Linux `sk_buff` structure) to handle packet buffers. While classic network stacks allocate such buffers and management structures dynamically, RTnet has to use a different scheme due to the real-time requirements. First, all `rt_skb`s are pre-allocated during set-up. As currently RTnet does not support buffer sharing between multiple users, the management structure and the payload buffer are forming a single memory fragment. And second, every `rt_skb` has a fixed size and can always carry the largest physical packet. This limitation is necessary to avoid shortages due to memory fragmentation and to allow exchanging of arbitrary `rt_skb`s between users.

Packet producers and consumers within RTnet have to create pools of `rt_skb`s in order to take part in the communication. During runtime, new `rt_skb`s are allocated from these pools. A reference in the `rt_skb` to its original pool allows to return it to its owner upon release. When a packet producer hands over a `rt_skb` to the destined consumer, the ownership changes only if the consumer is able to provide a free compensation `rt_skb` from its own pool.

Otherwise the packet is dropped, and the related buffer can immediately be reused.

Typical producers and consumers are the adapter drivers on one side and the sockets on the other. But also VNICs or management protocols like RTcfg and ICMP provide their own pools. Pools are created or resized in non-real-time context using the indeterministic memory allocation service of the underlying operating system. In order to allow socket creation and pool extension also in real-time context, the required `rt_skb`s are transferred in that case from a special global pool of preallocated buffers that has been created during the stack initialisation.

2.2 UDP/IP Implementation

Compared to a standard UDP/IP stack, several modifications were required to create the deterministic variant contained in RTnet. First, the dynamic Address Resolution Protocol (ARP) was converted into a static mechanism which is executed during the set-up. If a destination address is later unknown, no resolution requests are issued but a transmission error is returned to the caller. Otherwise, the worst case transmission latency of a packet would include the delay of a potential address resolution.

Second, the routing process was simplified. The output routing tables were optimised for the limited amount of entries used with RTnet. To accelerate the packet set-up, the tables also include the ARP results, i.e. the destination hardware addresses.

The defragmentation of IP packets needs special attention. In classic network stacks, this task is performed by the IP layer before any higher layers like UDP are involved. Thus, as the actual receiver is yet unknown, a global `rt_skb` pool is required for buffering all fragments before the last one has arrived. The addition of new fragments to an existing chain demands a lookup in the global list of all currently pending IP packets chains. Furthermore, incomplete chains have to be cleaned up after a timeout to avoid buffer shortages and to keep the global IP fragment list small.

The UDP/IP stack of RTnet contains several mechanisms to confine the effects of the defragmentation as far as possible to the receiving socket. For this purpose, the first fragment is used to immediately resolve the destination socket using an extended interface to layer 4. This information is then stored together with the fragment in a collector data structure. Further fragments are identified as usual by their IP addresses and IDs. To allow an efficient implementation of the collector, incoming fragments have to arrive in a strictly ascending order, otherwise the whole chain is dropped. Incomplete chains are cleaned up when the related socket is closed. The total number of collectors is limited in order to be able to specify an upper bound for the lookup latency.

2.3 Driver Layer

Network interface cards (NIC) are attached to the stack core using a Linux-like driver interface. This allows

straightforward porting of standard Linux drivers to RTnet, which has already been performed for about ten widely-used NICs. The NIC initialisation, configuration, and shutdown is still performed in non-real-time context under RTnet; porting standard drivers only requires to use the appropriate synchronisation mechanisms of the underlying RTOS here. However, special care has to be paid on the time-critical reception and transmission paths. They have to be audited in order to detect and avoid potential long delays while accessing the hardware.

A few extensions compared to the standard driver model are required to provide accurate timestamp services. RTnet does not depend on built-in timestamp clocks of the NIC, which are still not commonly available. Instead, the driver has to provide the packet reception and transmission time as precise as feasible. This means that the reception timestamp has to be taken for every packet right at beginning of the interrupt handler called upon the arrival. Furthermore, the driver has to provide the functionality to store the current time in an outgoing packet and trigger its transmission atomically. These measures widely reduce packet timestamp jitter to the single interrupt jitter which characterises platform and RTOS.

The driver layer furthermore provides two per-device hooks for redirecting transmission requests and MTU (maximum transmission unit) queries. Both hooks are transparent to the drivers. The transmission hook is used by the media access control layer RTmac and the capturing extension RTcap for managing, respectively, analysing outgoing packets. While standard network stacks typically provide only static device MTUs, RTnet offers logical channels of variable size up to the physical MTU to higher layers. The RTmac discipline TDMA utilises these channels to enforce specific slot sizes (see Section 3.2).

2.4 Application Programming Interface

Application programs can attach to the RTnet real-time services via a widely POSIX-conforming socket and I/O interface. The socket interface offers UDP and packet sockets for exchanging user data deterministically. The I/O interfaces provides access to additional features that services like TDMA (see Section 3.2) exports to users, for example clock synchronisation. Just as RTAI, RTnet permits both the classic kernel mode and more convenient user mode usage (Linux processes) of the API.

The related socket and I/O API functions are part of a separate interface concept called Real-Time Driver Model (RTDM). This interface addresses the specific requirements when accessing hardware on a mixed real-time system like Linux/RTAI, for instance differentiation between real-time and non-real-time service invocation. Currently, an implementation of RTDM comes with RTnet, but plans exist to merge the functionality into the RTAI project. This would also enable to utilise RTDM for other real-time devices drivers beyond RTnet.

2.5 Capturing Extension

A powerful extension of the RTnet core is the RTcap plug-in. It acts as a standard traffic capturing service for both incoming and outgoing packets over real-time NICs. Arriving packets are recorded together with a reliable high precision timestamp, solely depending on the interrupt jitter of the capturing system. RTcap adds only a small bounded overhead to the time-critical data paths when being installed on an active RTnet node. It furthermore cannot starve out any other packet user with respect to `rtskbs` because it maintains separate buffer pools for captured packets.

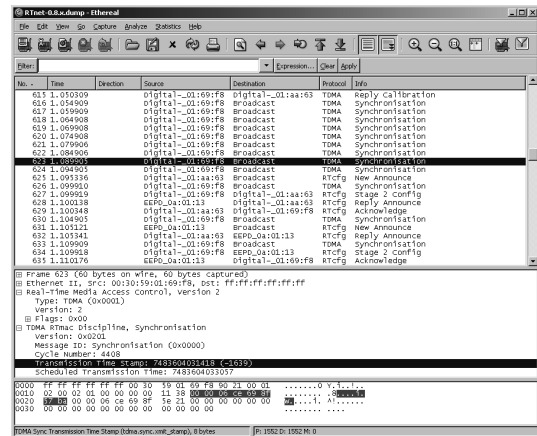


Figure 2. Using Ethereal with RTnet

Normal analysis network tools can be used with RTcap because a pseudo, read-only network device is created for every real-time NIC to forward the captured packets. Especially Ethereal [5], shown in Figure 2, is well-suited to dissect real-time communication as it fully understands the RTnet protocols. But the usage of RTcap in combination with traffic analyser is, of course, not limited to RTnet-managed networks or Ethernet. In principle, any transport media with RTnet-enabled drivers can be studied with RTcap's high timestamp accuracy.

3 Real-Time Media Access Control

As important as a real-time-capable stack implementation is a deterministic communication media. For instance, standard Ethernet, so far RTnet's primary media, does not provide adequate Quality of Service (QoS) features for hard real-time applications. Unpredictable collisions in hub-based Ethernet segments prevent short deterministic transmission times. Switches can overcome this issue but suffer from the risk of congestions which lead to packet delays or drops. QoS-enabled switches according to IEEE 802.1q are partly improving this situation, but they still require a centralised cabling which is often too costly for industrial applications.

Also other shared communication media may demand additional control over outgoing traffic in order to translate QoS parameters to a media-specific scheme or to ex-

tend existing QoS features where necessary. RTnet addresses the demand for deterministic and flexible media access control (MAC) mechanisms with its RTmac layer as described in the following. Moreover, as an example of a MAC discipline which is pluggable into the RTmac interface, a TDMA-based protocol is presented.

3.1 Pluggable MAC Layer

The RTmac is an optional extension to the RTnet stack. Although the stack is already functional without RTmac, it becomes mandatory if an underlying communication media lacks a deterministic access protocol. The RTmac layer was designed to provide these four elementary services to arbitrary software-based MAC implementations, here called disciplines:

- Interception of the crucial packet output path and redirection to discipline-specific handlers. For transmitting packets, this is performed right before the packet is passed to the NIC driver. Furthermore, a handler to override the device MTU on a per-packet basis can be installed.
- Exchanging discipline-defined control or data messages in a RTmac frame aside any user protocols.
- Discipline management on a per-device basis. To every real-time NIC, an individual MAC discipline can be assigned when it was registered with the RTmac layer.
- Packet tunnelling service for time-uncritical data as generated or received by the non-real-time network stack. This service creates a virtual network device for every RTmac-managed real-time NIC. Tunnelled packets are encapsulated by the RTmac protocol frame to distinguish between otherwise identical real-time and non-real-time protocols like UDP.

3.2 TDMA Discipline

Primarily for the use with standard Ethernet, RTnet provides a timeslot-based MAC discipline called TDMA (Time Division Multiple Access). TDMA in its current revision 2 is a master-slave protocol. It synchronises the clocks of RTnet nodes within a network segment. Furthermore, it defines the transmission time of any payload packet relative to synchronisation messages the master issues periodically.

A TDMA slave node can join a running network segment at any time provided it knows at least one parameter set of its slots. This set can either be configured statically or distributed via the RTcfg protocol (see Section 4). Given these parameters, the slave starts to join by sending a calibration request to the master. The master, in turn, replies with a message that contains the request arrival and reply departure times, both as precise as the system allows (see also Section 2.3). By taking its local departure and arrival times into account, the slave is able to calculate the

packet round-trip delay. This procedure is repeated over a certain interval in order to estimate the medium time t_{travel} between starting to transmit a packet on the master and gaining its reception time on the slave.

$$t_{travel} = \frac{1}{2n} \sum_{i=1}^n T_{recv,i}^{slave} - T_{xmit,i}^{slave} - (T_{xmit,i}^{master} - T_{recv,i}^{master}) \quad (1)$$

The master's synchronisation message contains the scheduled transmission time T_{sched} together with the timestamp taken right before packet release. This permits the slave to compensate potential scheduling jitters on the master node when calculating t_{offset} , the offset between local and global system clock. The slave can furthermore improve the precision of its own slot starting times T_{slot} .

$$t_{offset} = T_{xmit}^{master} + t_{travel} - T_{recv}^{slave} \quad (2)$$

$$T_{slot} = T_{sched} + t_{slot} - t_{offset} \quad (3)$$

Time slots can be freely arranged within an elementary TDMA cycle as depicted in Figure 3. Besides node assignment and offset, also the slot size can be defined within physical limits of the transport media. TDMA allows that a node uses multiple time slots per cycle. Furthermore, it is possible to set custom periodicity and phasing of a slot to limit the network load or to share slots between different nodes. A management tool is available under Linux to create and maintain individual configurations based on scripts. Even a runtime reconfiguration within certain constraints is feasible.

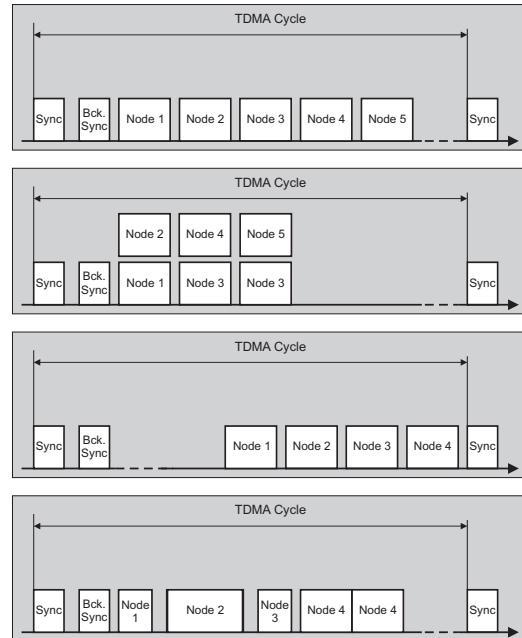


Figure 3. Flexible TDMA Slot Setup

In case multiple packets have been queued on a slot, the transmission order is defined by their priorities which can be set by real-time applications or RTnet services for

each message. 31 real-time levels are available, the 32nd and lowest one is reserved for time-uncritical data, i.e. VNIC traffic. With multiple slots per node, the need for a scheduling scheme arises. For efficiency reasons, TDMA provides explicit scheduling only. Slots are numbered on each node with ID 0 predefined for default real-time and ID 1 for non-real-time traffic. In case only a single slot is available, ID 1 is mapped on slot 0. Any additional slots are reserved for explicit assignment to arbitrary real-time applications via the socket API.

As the master is a single point of failure, its services can be backed up by one or more secondary masters. An additional time slot has to be assigned to every such backup master, marked as “Bck. Slot” in Figure 3. In case the primary master fails to transmit a synchronisation message, the next backup master on the time axis will start issuing its own messages. The offset between primary and secondary master is automatically compensated with a now larger difference between scheduled and actual transmission time contained in every synchronisation frame. When the primary master has been fixed and starts taking over again, it first synchronises its own clock on the active backup master in order to avoid significant clock skews. Afterwards it issues its own synchronisation messages again, and the backup master switches to stand-by.

The TDMA discipline creates a RTDM I/O device for every controlled network device. These I/O devices can be used to retrieve the clock offset introduced above and to synchronise a real-time task on the TDMA cycle.

4 Real-Time Configuration Service

During the revision of the first TDMA protocol it became apparent that a clear separation between RTmac disciplines on the one side and generic configuration as well as monitoring services on the other is essential for RTnet’s extensibility. For this reason, the Real-Time Configuration Service RTcfg has been designed in a discipline- and media-agnostic manner. It does not depend on a specific communication media given that broadcast transmissions are supported. The IPv4 protocol is supported but not mandatory. Other network protocols like IPv6 can be integrated, and physical addresses may be used even purely. The concrete tasks of RTcfg are:

- Distribution of essential discipline configuration data to newly joining nodes. This information is issued unsolicited, thus enabling nodes to join real-time networks on-the-fly as far as physical media and RTmac discipline allow.
- Monitoring of active nodes and exchange of their physical and logical addresses. This service can be used, for example, to set up and maintain the static ARP tables mentioned in Section 2.2. It is furthermore possible to build real-time network monitoring tools on top of RTcfg’s interfaces.

- Synchronisation of the real-time network start-up procedure. Specific RTmac disciplines or certain application scenarios may require common rendezvous points in order switch network mode or start applications synchronously.
- Distribution of arbitrary configuration data, even in the absence of TCP/IP with its typically used file transfer protocols like TFTP/FTP etc.

RTcfg is based on a client-server protocol. A central configuration server stores parameter sets of every managed client in a network segment. This information is used by the server to continuously invite any known but yet inactive client to join. The client’s start-up procedure as shown in Figure 4 consists of three stages. The first stage is completed after the client has received its single packet of initial parameters that is identifiable either through the physical or logical destination address. These parameters typically contain the minimum information required to set up a possible RTmac discipline, for example at least one TDMA slot configuration.

In the second stage after completing the discipline initialisation, the client announces its presence to any other network nodes which can then update their address information like static ARP tables. Already active clients reply on this announcement by sending the new node their own identification. The server replies in contrast by transmitting an optional second set of configuration data which can be scattered over multiple packets. After the server has received the final stage 2 acknowledge message from the last missing client node, the network is ready for a potential common operating mode switch in case such synchronisation is required.

As stage 3, an optional second rendezvous point is provided to both server and clients. It can be utilised to wait for all nodes to complete processing the configuration data they received during stage 2.

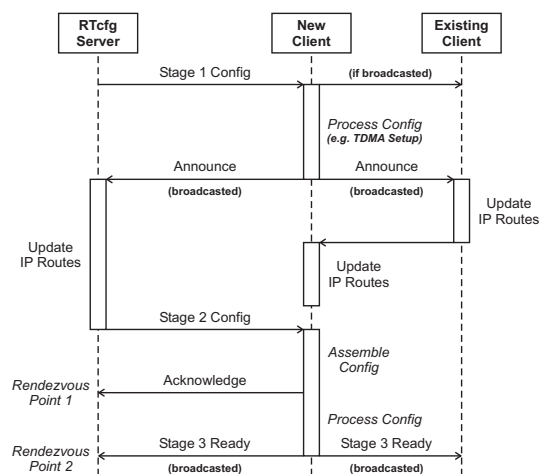


Figure 4. RTcfg Client Start-up in 3 Stages

After the setup completion the clients can be instructed to transmit low-frequent heartbeat frame to the server in

order to track potential node failures. If the server detects lacking heartbeat frames, it declares the client dead by broadcasting a related message to the remaining nodes. As a result, all nodes will remove any address of the broken client from their local tables. This enables a restart procedure of the repaired or replaced node. A failing RTcfg server can also be restarted, even on a different system, without the need to go through the full start-up procedure of every running node once again.

5 Integration of FireWire

FireWire, also known as IEEE 1394 [8], is a high-performance serial bus for connecting heterogeneous devices. Though firstly targeted for consumer-electronic applications, such as high-speed video transmission, many of FireWire's features make it well fit industrial and laboratory context. In the following subsections, an overview of FireWire is given and the current status of its integration to RTnet is described.

5.1 FireWire Overview

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 1394a specification. In 1394b specification, the speed even rises to 3.2 Gbps. Two types of data transaction are supported on FireWire: asynchronous and isochronous. As illustrated in Figure 5, a mix of isochronous and asynchronous transaction is performed by sharing the overall bus bandwidth, of which the allocation is based on 125 μ s intervals, so called FireWire cycles.

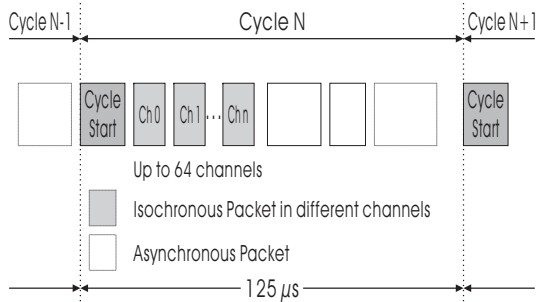


Figure 5. FireWire Cycle

Isochronous transaction targets one or more nodes by being associated with a multicasting channel number. There can be maximally 64 channels in total. Once bus bandwidth has been allocated for an isochronous transaction, the associated channel can receive a guaranteed time-slice during each 125 μ s cycle. Up to 80% (100 μ s) of each bus cycle can be allocated to isochronous channels. Because this transaction type does not re-transmit broken packets, but deliver data at constant, real-time 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 high-order 16 bits of address are used to identify nodes¹. An asynchronous transaction is split into two sub-transactions: 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 acknowledgement, asynchronous transaction is targeted for non-error-tolerant applications, like command and control message transmission in distributed control system.

Bus management on FireWire includes different responsibilities that can be distributed among one or more nodes: Cycle Master, Isochronous Resource Manager and Bus Manager. The Cycle Master broadcasts a start message at the beginning of each cycle. The Isochronous Resource Manager takes care of the allocation of bus bandwidth and isochronous channels. The Bus Manager has several functionalities including publishing the bus speed map and the bus topology map. Since FireWire connects devices that may not support the same top speed of data transmission, the bus speed map is used by a certain node to determine at what speed it can communicate with another node. The topology map may be used by end-users to optimise the bus topology for a highest throughput.

5.2 FireWire Stack and Connection to RTnet

The FireWire stack, as shown in Figure 6, is adapted from the Linux variant[9]. Functions in the kernel are decoupled into several modules. Application on the stack acquires either a portion of bus address or one or more multicasting channels, by using the primitives from the Application Interface and Management layer.

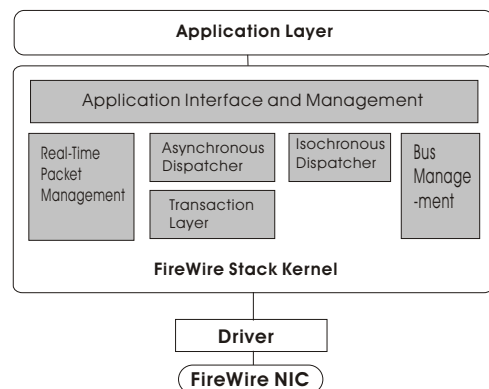


Figure 6. FireWire Stack

The RTnet mechanism for real-time packet management is applied to the FireWire stack as well. Both NIC driver and high-level applications are potential producers and consumers of packets. All packets are carried by a generic packet buffer structure `rtpkb`. Like in RTnet, pre-allocation of `rtpkbs` is done during set-up, with

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

each `rtpkb` carrying a fixed size of payload that is large enough to meet various scenarios.

The path of delivering incoming packets to application layer is realised by a real-time task, the so-called tasklet server. Upon arrival of a new packet, a suitable processing routine, either for asynchronous or isochronous, is hooked to the server as a tasklet. The server works under the rule First In First Served (FIFS), which means the packets are processed in the order of arrival time. When no tasklet is being queued, the server stays in idleness until the next packet arrives. A RTOS semaphore is used for the synchronisation between server and tasklet queue. Like the stack manager in RTnet, the server runs at a higher priority than application tasks.

The connection between FireWire stack and RTnet core is implemented through Ethernet emulation. The emulation is a module on application layer, using a portion of bus address to employ a protocol converting FireWire packets to Ethernet packets and vice versa. By using Ethernet emulation, FireWire functions the same as other real-time Ethernet devices in RTnet.

6 Application Protocols and Frameworks

The advantage that RTnet provides its real-time communication services through a widely standardised API instead of, for example, a specialised, solely fieldbus-oriented interface becomes obvious when considering application protocol layers. This section introduces some of them and also presents an exemplary concept for mapping an existing fieldbus middleware, CANopen, on RTnet's services.

6.1 netRPC – Remote Real-Time Procedure Calling

One of the first user of RTnet was its primary real-time execution platform itself. RTAI (3.x series) [17] comes with a plug-in called netRPC that enables a distributed usage of its RTOS services. This remote procedure calling service (RPC) is built upon the UDP/IP protocol. It can either be attached to the Linux non-real-time network stack, typically for testing and demonstration purposes, or to the RTnet API. In the latter case distributed hard real-time is provided to the RTAI applications almost transparently. Some of the RTAI developers make use of this feature in their real-time multi-body dynamics analysis software MBDyn [13].

6.2 RTPS Protocol

The Real-Time Publish-Subscribe Protocol (RTPS) [14] has been developed in order to provide real-time communication services over unreliable IP networks like Ethernet. The protocol contains mechanisms to detect critical packet delays or losses and avoids indeterministic re-transmissions, as for example TCP causes, by using UDP as transport protocol. In order to keep real-time communication operational on Ethernet, only a low network load is

acceptable in RTPS segments. RTPS is available as a commercial product (NDDS) and is included in various industrial products, for instance in certain Schneider PLCs.

Moreover, an Open Source implementation of RTPS called ORTE [2] is available. ORTE runs on a large number of platforms over conventional UDP/IP stacks and, additionally, supports RTnet on top of RTAI. By utilising RTnet's hard real-time UDP/IP services, RTPS can now be used even under high non-real-time network load, as RTnet reliably separates this traffic from the time-critical data.

6.3 Real-Time Control Frameworks

Both for research and industrial scenarios, increasingly complex control tasks demand powerful frameworks to facilitate the development of distributed real-time systems. One of such frameworks has been developed at the Real-Time Systems Group in Hannover with the focus on robotic research [20]. This framework transparently supports distributed applications both deterministically over RTnet (UDP/IP) and without timing guarantees over standard TCP/IP. Its communication models include remote procedure calling as well as producer-consumer schemes.

A similar framework, OROCOS, also makes use of RTnet for closed-loop control [15]. Moreover, plans exist for OROCOS and the related OCEAN project to run RT-CORBA over RTnet. The latter project already evaluated an earlier version of RTnet and concluded that integrating it as pluggable protocol into the RT-CORBA implementation ACE/TAO is a promising approach [19].

6.4 CANopen

The CAN in Automation organisation has developed CANopen as an application protocol and device model for the automation domain [1]. Beyond its original use on top of the CAN fieldbus, CANopen has recently been adopted by two commercial real-time Ethernet solutions, ETHERNET Powerlink [3] and EtherCAT [4]. Both approaches are, as well as RTnet, quite different compared to the CAN bus with respect to node addressing, message priorities, or communication models. Therefore, ETHERNET Powerlink and Ethercat only reuse the device profiles specified by CANopen. In following, the feasibility and potential of adopting CANopen to RTnet is briefly analysed. Such an extension would enable classic automation applications like soft-PLCs to run more straightly over RTnet.

As CAN itself is agnostic to message source and destination addresses, CANopen maps the common three addressing modes broadcast, unicast, and multicast on CAN message identifiers. Broadcast messages are used for network management, synchronisation, time stamping, and alarming purposes. CANopen exchanges so called Service Data Objects (SDO) for time-uncritical direct communication between two nodes as unicast messages. Process Data Object carrying the real-time data are transmitted according to the multicast scheme with a single producer and an arbitrary number of consumers.

RTnet supports broadcast as well as unicast both via UDP and user-defined Ethernet protocols. As multicast support is not yet part of RTnet, such messages can be issued transitionally either via unicast in case only a single consumer exists or as broadcasts using additional software filters on the receiving nodes. Basically, an extension of the Communication Object ID (COB-ID) format is required, which was originally defined with solely CAN IDs in mind. While CAN prioritise messages implicitly according to their ID, an explicit value is now required which also encode the output channel on RTnet. An extended COB-ID would demand the following fields:

- ID type (UDP/IPv4, UDP/IPv6, Ethernet, CAN, etc.)
- Destination node address (IP, Ethernet MAC, etc.)
- Message ID (UDP destination port, Ethernet frame type, CAN ID, etc.)
- Priority and channel (RTmac queuing priority, TDMA slot, etc.)

The CAN-specific Remote Transmission Requests (RTR) are utilised by consumers for soliciting a PDO from the producer. This protocol can be emulated by sending an empty PDO with identical COB-ID to the producer.

Based on the proposed addressing scheme, typical CANopen stacks, for instance one of the various free implementations [6], may already be reused on top of RTnet. Certain CANopen services could be mapped directly on RTnet equivalents. RTcfg provides heartbeat mechanism which can replace CANopen's variant. TDMA comes with an API to synchronise nodes and distribute a common time base, services that be used in place of the CANopen protocol. Additional optimisation potential lies in larger transfer fragments when exchanging SDOs. CANopen's limitation to CAN-related 8 bytes can be easily overcome by defining new, COB-ID-specific SDO upload and download protocols that make use of different maximum packet sizes (e.g. almost 64 KB via UDP/IP).

7 Summary and Outlook

This paper introduced RTnet as an adaptable and extensible framework for deterministic communication over standard Ethernet, FireWire, or other suited media. Its open, standard-oriented, and modularised structure allows numerous application scenarios like distributed real-time systems, fieldbus coupling devices, intelligent I/O interfaces, low-cost real-time network analysers, etc. Application software may either interact directly with the RTnet API, or middlewares like RTPS or CANopen can be build over RTnet's services.

Future work will focus on further integration of FireWire, new media like Gigabit Ethernet, and interoperation with additional middlewares. To decouple organisational dependencies, the RT-FireWire stack has recently become a separate project. Based on the connection to

RTnet via Ethernet emulation, the adoption of FireWire's transaction modes and clock synchronisation for RTnet services will now be addressed. Furthermore, the potential of layering CANopen over RTnet will be analysed and can lead to the implementation of an extended CANopen stack.

The current RTnet implementation has been build upon free software, it tightly interacts with many Open Source projects, and it is therefore available under Open Source licenses, too. For downloads and further information, visit

www.rts.uni-hannover.de/rtnet

References

- [1] CiA. *CANopen, Application Layer and Communication Profile*. CAN in Automation, Feb. 2002.
- [2] O. Dolejs, P. Smolik, and Z. Hanzalek. On the Ethernet use for real-time publish-subscribe based applications. In *5th IEEE International Workshop on Factory Communication Systems*, Vienna, Austria, Sep. 2004.
- [3] ETHERNET Powerlink Standardization Group. www.ethernet-powerlink.org.
- [4] EtherCAT Technology Group. www.ethercat.org.
- [5] Etherreal. www.etherreal.com.
- [6] CANopen free software resource center. canopen.sourceforge.net.
- [7] F. T. Y. Hanssen and P. G. Jansen. Real-time communication protocols: an overview. Technical Report TR-CTIT-03-49, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, Oct. 2003.
- [8] IEEE. *IEEE standard for a high performance serial bus, Std 1394-1995 and amendments*, 2002.
- [9] IEEE 1394 for Linux. www.linux1394.org.
- [10] LinuxDevices.com. Lineo announces GPL real-time networking for Linux: RTnet. www.linuxdevices.com/news/NS4023517008.html, July 2000.
- [11] J. Loeser and H. Haertig. Low-latency hard real-time communication over switched Ethernet. In *16th Euromicro Conference on Real-Time Systems*, Catania, Italy, 2004.
- [12] J. Martínez, M. Harbour, and G. J.J. A multipoint communication protocol based on Ethernet for analyzable distributed real-time applications. In *2nd International Workshop on Real-Time LANs in the Internet Age*, 2003.
- [13] Multibody dynamics analysis software on real time distributed systems. www.aero.polimi.it/~mbdyn/mbdyn-rt.
- [14] Real-Time Innovations, Inc. *Real-Time Publish-Subscribe Wire Protocol Specification, Protocol Version 1.0, Draft Document Version 1.17*, 2002.
- [15] Open Robot Control Software. www.orocos.org.
- [16] P. Pedreiras, L. Almeida, and P. Gai. The FTT-Ethernet protocol: Merging flexibility, timeliness and efficiency. In *14th Euromicro Conference on Real-Time Systems*, 2002.
- [17] Real Time Application Interface. www.rtai.org.
- [18] J. Schwager. Real-time Ethernet in industry automation. www.realtime-ethernet.de.
- [19] M. Wild et al. OCEAN deliverable D2.1: Design of the DCRF lower layers including hardware requirements. www.fidia.it/download/ricerca/ocean/deliverable2.1.pdf, 2003.
- [20] O. Wulf, J. Kiszka, and B. Wagner. A compact software framework for distributed real-time computing. In *5th Real-Time Linux Workshop*, Valencia, Spain, Nov. 2003.