Technical Report

A Complex Protocol Layer as a linux User-Space Process

António Barros

Filipe Pacheco

Luis Miguel Pinho

TR-061006

Version: 1.0

Date: October 2006

A Complex Protocol Layer as a Linux User-Space Process

António BARROS, Filipe PACHECO, Luís Miguel PINHO

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: {amb, ffp, lpinho }@dei.isep.ipp.pt

http://www.hurray.isep.ipp.pt

Abstract

With the current complexity of communication protocols, implementing its layers totally in the kernel of the operating system is too cumbersome, and it does not allow to fully use the capabilities which are only available in user space processes. However, building protocols as user space processes must not impair the responsiveness of the communication. Therefore, in this paper we present a layer of a communication protocol, which, due to its complexity, was implemented in a user space process. This layer must communicate with the lower layers of the protocol, which for responsiveness issues are implemented in the kernel. This protocol was developed to support large-scale power-line communication (PLC) with timing requirements.

A Complex Protocol Layer as a linux User-Space Process

António Barros, Filipe Pacheco, Luis Miguel Pinho

Department of Computer Engineering, ISEP, Polytechnic Institute of Porto,
Rua Dr. António Bernardino Almeida, 431, 4200-072 Porto, Portugal
{abarros, ffp, lpinho}@dei.isep.ipp.pt

Abstract

With the current complexity of communication protocols, implementing its layers totally in the kernel of the operating system is too cumbersome, and it does not allow use of the capabilities only available in user space processes. However, building protocols as user space processes must not impair the responsiveness of the communication. Therefore, in this paper we present a layer of a communication protocol, which, due to its complexity, was implemented in a user space process. Lower layers of the protocol are, for responsiveness issues, implemented in the kernel. This protocol was developed to support large-scale power-line communication (PLC) with timing requirements.

1. Introduction

Using power lines for communication is not a new idea [1]. Use a already deployed infrastructure eases communication costs, not only for energy providers, but also for user applications. Nonetheless, communication technology must be adapted to this peculiar medium [2].

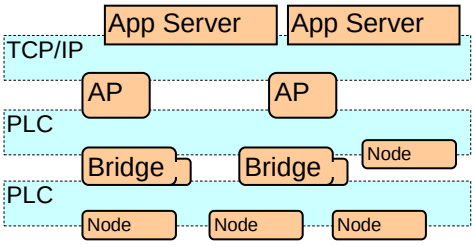

Fig. 1 – General REMPLI Architecture

One of the goals of the REMPLI (Real-time Energy Management over Power-Lines and Internet) project [3] is to implement an infrastructure (Fig. 1) for real-time communication, in order to remotely access monitoring and control equipment [2]. Within the power lines, a two-level hierarchical system is used. New protocols where developed with special consideration given to path failure management, redundant paths, data fragmentation and real-time traffic processing using knowledge from MANETs (Mobile Ad-hoc Networks) and Wireless Networks [5] research.

In the REMPLI architecture, the end-to-end application services are provided over an already existing master-slave Medium Access Control (MAC) [4]. The base network only encompasses a single PLC network level, and so the implemented system provides the end-to-end communication and routing services.

This paper presents the implementation challenges of the protocol layer and it is organized has follows. In the next section, we present the services already provided by the base network. Then, in section 3, we summarize the architecture of the protocol layer that is used for queuing and routing within the PLC system. Section 4 presents the target hardware platform and section 5 details the implementation aspects of the Transport Layer in this platform. Finally, section 6 presents the status of the implementation and the planned test strategies.

2. The REMPLI Network Layer Services

2.1. The Power Line Communication System

Power line grids divided in several voltage groups: higher voltage for power transportation over large distances, and lower voltages for power distribution on smaller areas. The REMPLI Project aims at the last two levels of this distribution grid. On the low-level end, we have the REMPLI Nodes. Between the two voltage levels, there is a transformer for voltage conversion (and power transmission) and a REMPLI Bridge for data forwarding. On the entry side of the medium voltage network, the REMPLI Access Point (AP) interconnects the PLC system and a broadband backbone (already available in most of these facilities).

For the device-to-device communication, the REMPLI system uses a time-division-multiplex Master/Slave communication protocol [4]. Requests are time-interleaved by the Network Layer (NL): a request is sent in one slot and the response is received *n* slots afterwards with *n* fixed in each network and with typical values of 3 or 4. Multi-master is also supported, for example: the first 2 time slots used by the first Master

1-4244-0777-X/06/$20.00 ©2006 IEEE

and the other 2 time slots for the second Master. The data payload of each slot is fixed and typical values for raw data payload (i.e. including the base network own headers) are 64 and 128 bytes. Increasing the data length decreases header overhead and increases the error probability, decreasing the length enables more messages per period.

2.2. Services available to Applications

From the point of view of applications that use the Transport Layer (Fig. 2), each AP will have direct connection with several devices (Bridges and Nodes).

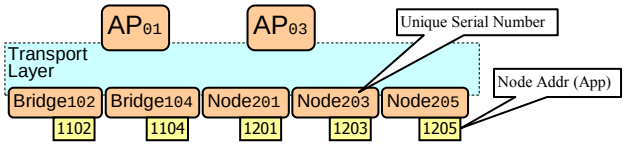

Fig. 2 – Application's view of the system

In this document, we use "Unique Serial Numbers" to identify each device on the text: i.e. "Node 203" means "Node with the Unique Serial Number 203". Applications calls use Node Addresses; a configuration table converts "Node Addresses" to "Unique Serial Number". In the example network of Fig. 2, applications in either AP may issue requests to any of the network devices. Delays and available bandwidth will vary for each AP – Device combination.

From the applications point of view there are only two classes of devices in the network: "Access Points" and "Bridges/Nodes". The main services that may be issued from the AP side are [2]: Unicast Request with response; Unicast Request without response. From the Node/Bridge side: Respond to Request; Send Alarm.

The Unicast Request without Response service sends data to a particular device without confirmation of delivery. The Unicast Request with Response service is similar but the node/bridge must issue a data response. The node/bridge uses the Response to Request service to send this data. Only one response may be issued for each request.

The Send Alarm service is issued by a node/bridge to deliver data to; at least, one of the available APs, this delivery will be confirmed to the node/bridge.

All the data services may transmit large data blocks (up to 32MB) although time-critical applications will usually use very small data blocks (up to 100 bytes).

3. The Transport Layer Architecture

The REMPLI Transport Layer is built by four inter-connected modules as presented in Fig 3.

The *RCI Manager* (RCIM) module distributes messages between the Queue Manager / Transport Route Manager and the Applications themselves using the REMPLI Communication Interface (RCI).

The *Network Layer Interface* (NLI) module distributes messages between the Queue Manager / Transport Route Manager and the NL doing parameter conversion when needed.

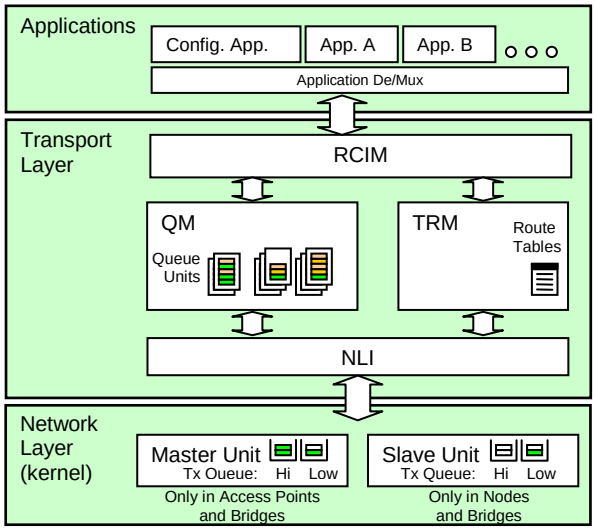

Fig. 3 – REMPLI internal architecture

The *Queue Manager* (QM) module manages all the packet data information. Tasks like queue generation, disposal, fragmentation and transport system header processing are done at this module. This module also multiplexes requests from the Applications and the Transport Route Manager to the Network Layer Interface data transmission services. All the header information needed for these services has been reduced to just 2 bytes. Since the usable data length of the NL is not an integer multiple of a fixed value Bridges have a complex task to forward data

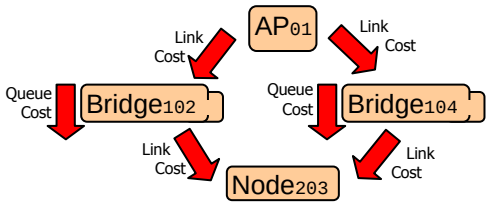

Fig. 4 – Route cost/delay calculation

The *Transport Route Manager* (TRM) module handles not only the scheduling tasks but also when internal Transport Layer communication is needed. TRM takes care of the system start-up (including device plug-and-play operation) and automatic route discovery. When a new data request is queued in the QM the TRM will produce a route (including a null route if no path available). The TRM will also instruct the QM when to served a queue. For simplicity all fragments of a particular request (and response, if applicable) are routed using the same path. Routing decisions are taken on the AP once per request. The AP scheduler estimates the cost of each route to make the routing decision (Fig 4) using AP/Bridge Link Transmission Cost; Bridge/Node Link Transmission Cost and Bridge queue status.

4. The HyNet System Hardware

The prototype REMPLI embedded processor board is based on the HyNet32XS [8]. This is a 32-bit RISC/DSP microprocessor that integrates components for Ethernet, USB 1.1, CAN-Bus, ATM and UART interfacing (including M-BUS metering devices). The DSP interfacing allows easy integration with the PLC modem, based on the single-chip DCL-2 PLC [7].

The board has 32 MB flash memory and 64 MB of DRAM. The Operating System is a processor-specific uClinux (www.uclinux.org) port, a lightweight UNIX version that is also available for other embedded devices. The software for this system is written in C language and is built and linked using a uClibc (www.uclibc.org) tool-chain ported specifically for this processor board. uClibc is a libc library, designed for embedded systems.

5. Embedded Software Implementation

The REMPLI Transport Layer (TL) is implemented as a multi-threaded single process. The TL is loaded to user-space during the boot procedure. The four major components described in section 3 – RCIM, QM, TRM and NLI – execute in their own thread, on the same process, allowing tighter integration. The behavior of these components was tested on a simulation system using OMNeT++ (www.omnetpp.org), a discrete-event simulation system for Windows and UNIX platforms.

5.1. TL Internal Messages

The four TL components interact exclusively by exchanging messages. So a high amount of internal traffic is expected. A message exchanged by two components can transport parameters and payload data. Parameters are used to select proper processing of the received message. Payload data is variable size data that should travel unmodified until its final destination.

Messages can take several different paths inside the TL, according to its kind and system status. A unique TL message structure was defined and each message contains the complete suite of parameters required. The structure also contains a pointer to, and the length of, the data payload. This fixed-size structure results in memory overhead since most parameters are unused in most messages, but this is rewarded by faster data processing and transport. The message structure is not moved nor resized during its lifetime: TL component use the *address* of the structure to access or change message parameters. A component can pass the changed message to another component without requiring new memory allocation of the message.

Messages flow between components via linked lists using dynamic memory. Each component has an associated receiving message indication list, working as a FIFO mailbox, as represented in Fig. 5. Components communicate by posting message indications on other components' mailbox.

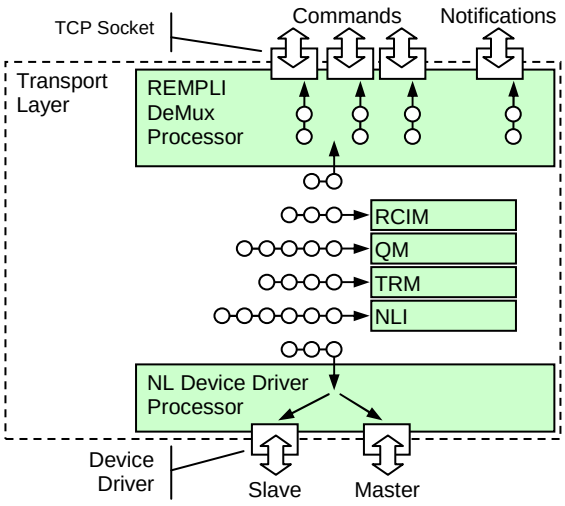

Fig. 5 – REMPLI TL architecture: components and message queues.

5.2. Message operability library

A library was developed for TL internal message operability including services to create, operate, transport, destruct and schedule messages.

Message creation involves allocating memory for a clean message header.

The library contains functionalities to define each parameter. Each 'set' functionality has a 'get' counterpart that allows a TL component to read message parameters and data.

Attaching a payload block to a message can be done in two different manners: by copy or by reference.

Copy is used when the data resides on a static location or the original data is to be preserved. The library allocates new memory and copies the data.

Reference is used if the payload data is already in a dynamic memory block and so the pointer is set directly.

In order to keep data integrity a payload block can only be part of one message; and once the payload block is attached to the message, it will be destroyed with the message (or freed when replaced by another payload).

Any TL component can collect messages from its respective mailbox and send messages to another component's mailbox. Data integrity is kept with one mutex per mailbox. The mailbox message collector and the message sender share a mutex. The component thread sleeps while the mailbox is empty. When the component gains access to its non-empty mailbox, one message is removed for processing and the component code executed. To send a message to another component mutex access is required prior to adding the message to the destination mailbox. Mailbox mutexes are locked for a very short time: the necessary time to remove an item or to add a new item. Components do not have direct access to mailbox mutexes.

When a message has accomplished its purpose, it should be eliminated. The memory allocated for the header and possible payload is then freed.

TL components may need to use timed messages to support timeout operations and repeated timed operations. Timed messages are set using a special message call. This call generates a new thread that will sleep for the defined period, after which will send the message back to the issuer's mailbox to be processed like any other message (including FIFO order).

5.3. Higher-level messages

The TL communicates with DeMux using TCP sockets. Each message is composed of a header, a payload and an end-of-block field. Message headers are defined in a unique fixed-size structure, and identify the nature of the message and the payload length. The payload is variable-length and treated like an opaque block by the TL in most situations.

Two kinds of messages are exchanged in the TCP connection: Commands and Notifications. Command messages are issued by the DeMux and must have a Success/Error reply from the TL. Several commands may be generated by the Demux before the respective replies are received. The TL also allows more than one TCP channel for Command/Reply messages. A pair of threads manages each channel at the TL side. One thread deals the incoming messages and translates them to the RCIM Mailbox. The other thread picks messages from the Demux Processor mailbox and sends the replies. Internal TL messages have a parameter used by the DeMux to match replies with commands and another parameter that identifies which TCP channel to use.

Notification messages are originated by the TL to report any event, or received data. A single channel is used for these messages. When a Notification message is to be delivered to the DeMux it will be handled by the Notification channel thread without any handshake mechanism.

5.4. Lower-level messages

The NL interface is specified as two Linux device drivers, one for Master NL and other for Slave NL.

An AP has one Master NL Linux device driver. A Node has also one Linux device driver, but with Slave-specific messages. For these devices, the TL employs two threads. One thread acquires data from the NL and generates a TL internal message in the NLI's mailbox. The second thread acts the opposite way, processing and delivering a message to the NL.

A REMPLI Bridge connects to two networks, so it requires two Linux device drivers: one for the Master and another for the Slave. Three threads are required in a Bridge: two acquire data from each of the device drivers and the third thread will distribute the outgoing data by the two device drivers, selecting the appropriate device according to the message type.

6. Implementation tests

Implementation tests are now being carried on a testbed located at iAd GmbH facilities. This testbed includes multiple prototype boards connected over a PLC network. There are alternative physical layouts of the testbed for direct-connection test and bridged network testing. Apart from the physical layout changes that have to be done on-site, all the debugging can be done remotely over SSL connection including powering the devices. A battery of tests is already defined to test the integration between the diverse layers of REMPLI and was run on the simulation system. Tests carried out until now show that the TL is working as expected in the final hardware. Issues solved included limitations on local stack and some restrictions on the C compiler.

7. Conclusions

An end-to-end communication system was devised over a master-slave MAC that overcomes the inherent dynamic topology of PLC. This system provides queuing and routing mechanism as well as other accessory services. This paper presented an implementation-oriented view of the mid-level protocol layer that is being implemented for a two-level PLC architecture.

The REMPLI project is now in the integration phase and field trials will begin soon.

References

[1] M. Lobashov, G. Pratl, T. Sauter, "Implications of Power-line Communication on Distributed Data Acquisition and Control System", *IEEE Conference on Emerging Technologies and Factory Automation*, Lisbon, Portugal, 2003.

[2] A. Treytl, T. Sauter, G. Bumiller, "Real-time Energy Management over Power-lines and Internet", *Symposium of Power-Line Communication and its Applications,* Zaragoza, Spain, 2004.

[3] Real-time Energy Management over Power-Lines and Internet, NNE5-2001-00825, http://www.rempli.org

[4] G. Bumiller, M. Sebeck, "Complete Power-Line Narrow Band System for Urban-Wide Communication", *Symposium of Power-Line Communication and its Applications*, Malmö, Sweden, 2001.

[5] F. Pacheco, L. M. Pinho, E. Tovar. "Queuing and Routing in a Hierarchical Powerline Communication System", *IEEE Conference on Emerging Technologies and Factory Automation*, Catania, Italy, 2005.

[6] D.B. Johnson, D.A. Maltz, Y-C. Hu, "The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR)", Internet Draft, IETF MANET Working Group, April 2003.

[7] Datasheet of the DLC-X family. iAd GmbH, 2001.

[8] Datasheet – HyNet32XS, Hyperstone AG, 2003.