NTNU
Norwegian University of
Science and Technology

Priority Based Message Stack

Steinar Lieng Fredriksen

Master of Science in Engineering Cybernetics
Submission date: June 2008
Supervisor: Amund Skavhaug, ITK
Co-supervisor: Trygve Lunheim, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

Nera has a system with moderate real time requirements in which message passing between separate parts of the system is facilitated by an Ethernet bus. An off-the-shelf general purpose TCP/IP stack is used for messaging in the system. In order to increase the level of determinism in the time domain the TCP/IP stack is to be replaced by a priority based message stack. A method for strict prioritizing between classes of messages is to be implemented.

The requirement is to construct a message management layer on top of existing Ethernet drivers to guarantee enforcement of assigned priority values for each message transported.
This layer should preferably mimic the industry standard BSD sockets API to the extent possible while still allowing individual priorities to be assigned. Protocol support will be limited to BOOTP, ARP and UDP which are all stateless unreliable datagram protocols. No stream or state handling will be needed.

Detailed requirements for the application are to be documented by the student in the form of a Requirements Specification. The subdivision of tasks is to be documented in the form of a project plan. The project is to be carried out as a standard software development cycle with requirement specification, design, implementation and testing.

The possibility of using COTS in the implementation is to be evaluated.

Assignment given: 07. January 2008
Supervisor: Amund Skavhaug, ITK

Preface

I wish to thank my supervisor Amund Skavhaug for being a great source of inspiration and a verbal whirlwind. I would also like to thank Trygve Lunheim for great insights into the general technologies that were available. Any errors are my own.

I would like to thank Tonje Fløystad and Dag-Erik Laumann at Nera for all their help with carrying out this project, and especially for keeping the requirements stable.

I would also like to thank my significant other, Anne-Lise, for all her patience and helpful comments during this project.

Abstract

To enable deterministic scheduling in a distributed embedded system an existing open source embedded TCP/IP stack has been modified to support strict priority queuing.

The embedded target system has a fully switched closed Ethernet backplane used for internal communication. The problem is that high-volume configuration file downloads interfere with low-volume internal signaling, such as alarms, status reports, performance data and other statistics. The network was already designed using a type of switches which could support several Quality of Service schemes. An open source embedded TCP/IP stack, lightweight IP (lwIP) was evaluated, and found to be a suitable foundation for the developed application; the Priority Based Message Stack (PBMS).

PBMS is a modification of lwIP with support for selective packet forwarding and reception based on the IPv4 Type-of-Service (TOS) field. Support for the BOOTP protocol, as well as a nonblocking send operation, was also implemented. PBMS is easily portable in the sense that it only requires a clearly defined basic set of generic OS functions in order to be ported to new systems. A generic interface to Ethernet drivers must also be developed.

The implementation was desktop tested on a Linux platform for correct functionality in the IP, UDP and BSD Socket modules. The performance of the priority scheme compared to a best-effort strategy was also measured. These tests showed that stack-internal strict prioritization based on the IPv4 TOS precedence bits have a clear potential for offering deterministic transfer times in Nera's distributed embedded system.

Without prioritization, with contention for the network link, an average output time of 480 microseconds was measured from the time the socket ***sendto*** function was called until the IP ***output*** function had been executed. Using strict prioritization under the same conditions this sequence of functions executed in only 15 microseconds in 99 percent of the test runs.

The proper prioritization within the TCP/IP stack is only one of several queuing points in the target network. The existing switches, along with the proper TOS marking of each packet, will ensure an unbroken chain of priority queueing all the way to the destination. Some network endpoints might have variable latencies, which will affect the round-trip time (RTT). A measurement methodology to determine this has been described.

A test methodology for round-trip and one-way delay time measurement will be used to validate PBMS in the target system. These tests can also be used to estimate the latencies in those network endpoints mentioned above. With this data at hand a worst-case RTT for the entire system can be calculated. The target system integration and validation will be carried out immediately after the completion of this thesis.

Table of contents

Figures

Tables

Terms

API	Application Programming Interface
ARP	Address Resolution Protocol
BOOTP	Boot Protocol
BSD	Berkeley Software Distribution
C module	Most of the code described in this project follows the idiom of having all functions and structs declared in a header file, and defining them in a .c-file with the same name.
COTS	(Software) Components off-the-shelf / Commercial off-the-shelf
CSMA/CD	Carrier Sense Multiple Access / Collision Detection
DSCP	Differentiated Services Code Point
IETF	Internet Engineering Task Force
IPPM	Internet Packet Performance Metric
lwIP	lightweight Internet Protocol
MTU	Maximum transmission unit, usually 1500 bytes.
OSS	Open Source Software
PBMS	Priority Based Message Stack
PCB	Protocol Control Block. A data structure which defines a connection in the context of a given protocol.
(platform) port	A collection of code written for a specific platform, e.g. a certain operating system, to enable the use of some other code on that specific platform.
QoS	Quality of Service
Refactoring	Modifying source code without changing the functionality it offers, often to enhance readability, modifiability and related attributes.
RFC	Request For Comments
RPC	Remote Procedure Call (generalized)
RTOS, OS	(Real Time) Operating System
RTT	Round trip delay time
SLOC	Source lines of code
TAP Interface	Virtual Ethernet device in Linux
TCP/IP Stack	Transport Control Protocol / Internet Protocol Stack. The term usually refers to a full networking stack. (Not limited to TCP over IP)
TOS	Type of Service
UDP	User Datagram Protocol
WCET	Worst-Case Execution Time

1 Introduction

This report describes a project that was carried out to develop a Priority Based Message Stack (PBMS). PBMS has been developed for the company Nera Networks AS, which in this report will be referred to simply as Nera.

This report is structured around the employed software development process and associated document deliveries. This may lead to an uncommon organization of the contents, so this introduction chapter will try to address any such issues. This chapter will also describe the delimitation of the project.

The problem that was to be solved in this project is essentially defined by three items:

1. The problem statement given at the beginning of this report
2. The requirement specification given in chapter 3
3. The project plan given in chapter 2

The project plan defines which documents Nera wanted to be delivered in addition to the actual implementation. These have been included in their entirety as chapters in this report since they were an integral part of the work that has been carried out. The requirement specification details both software requirements and process requirements, such as which programming language to use, which protocols the Priority Based Message Stack (PBMS) must support and the performance requirements it must meet.

Code

The implementation of PBMS is built on lwIP which comprises over a hundred source files, so the implementation is delivered in electronic form only. Selected code segments are included to demonstrate certain features, but in general one must have access to the files accompanying the report to use PBMS.

Whole blocks of code are shown with `the courier font, size 9`. Function and variable names discussed within normal paragraphs are shown with the ***normal font in bold and italic,*** so that they are easily distinguished from *quotes, which are in italic.*

A note on scope and terminology

This report is written for an audience which is familiar with the problem domain.

The terminology on software architecture and software components (COTS) is from the book "Software Architecture in Practice" by Bass, Clements and Kazman [1]. Since that book was used as the curriculum in a course I took, *TDT4240 Software Architecture*, I have assumed that this terminology is well known. The definitions of all such terms can be found in [2].

Wiki and mailing list sources

The lwIP documentation is maintained at a wiki by a community of developers. This information is generally not considered to be academically reliable. Such information will not be used to verify anything, unless it can be corroborated by other sources. The same goes for any information found in a mailing list, which is even less reliable than a wiki.

1.1 The report chapters

The rest of this introductory chapter comprises the project delimitation, a description of the target system, and the problem PBMS is designed to avoid. Chapter 2 is the above mentioned project plan, which is presented in the original form. A discussion of how the process was carried out, along with the actual progression of individual tasks is presented in chapter 11.

The theoretical foundation for the implemented design is presented in chapter 6.4 – the design rationale.

Chapter 3 is the PBMS user documentation which contains general introductions to lwIP and PBMS. That presentation precludes the discussion of why lwIP was chosen over other possible solutions, which can be found in chapter 5. Chapter 5 also contains a general discussion of component based software development, along with a description of the software component selection process carried out in this project. An evaluation of this process is given along with the evaluation of the project plan in chapter 11. General project experiences are also presented there.

The software architecture of PBMS is documented in chapter 6, while the design and implementation is documented in chapter 7. An overview of the different tests that are to verify PBMS according to the requirements are given in chapter 8 and a report that details the results from these tests is in chapter 9. The general project results are presented in chapter 10, these are then discussed in chapter 11, and the project conclusion is finally presented in chapter 12.

1.2 Project delimitation

Upon the completion of this project I will work at Nera's facilities to implement the required port for the Integrity RTOS. All tests on the embedded target will be performed after the port has been implemented. Some test procedures, such as one-way delay and round-trip delay measurement, have thus only been described in this report, not carried out. Any outstanding tasks described below may also be finished during that period.

1.2.1 Implementation

Most of the desired functionality has been implemented, either through coding or reusing available components. The BOOTP protocol support is not complete, what remains is to transmit the proper reply upon the reception of a valid request. Implementing this should be quite simple; the only reason it is not already finished is a lack of time. I chose to focus on testing the determinism of the scheduling policy as much as possible, since this essentially is the purpose of the entire stack. It is also not yet decided if the configurable ARP update on ingress IP is desirable in the target system.

1.2.2 Test

Desktop tests have been performed for most of the "normal operation", and a few error conditions have been triggered and analyzed, especially related to full message buffers. Several more error conditions should be deliberately triggered before PBMS can be put to use in the target system.

The socket interface has not been tested thoroughly for user errors, but this is something that will be done before PBMS is eventually put into use. The PBMS interface must also be integrated with the target OS RPC mechanism. A thorough evaluation of the quality features of the solution, mainly performance, modifiability and dependability, will also be important when deciding if PBMS is to be deployed in the target system.

The test platform was a single desktop Linux PC using hardware emulation only, specifically the TAP interface. A measured round trip time on such a platform would not be a good indicator of the target system performance. Therefore only internal processing time has been measured during this project.

The necessary drivers for running lwIP on a Linux test platform were already available along with the lwIP source, but it did not contain a proper Ethernet device driver. To compose a physically distributed test system would require an implementation of a proper device driver, as well as replacing the built-in Linux TCP/IP stack with PBMS. It would also be necessary to use switches similar to those in the target system. This was not done, since it would probably require too much effort compared to the value of the test results.

1.3 Target system description

Some system details are intentionally left out of this description, since Nera did not want them published. Details about the target system are mainly used for evaluating the proposed design and implemented solution, but generic descriptions should suffice for this. Knowing the exact type of certain network elements should not be necessary.

The target system is presented in this way so that the report and source code could be made available for the academic community immediately after its completion. The alternative could have been to restrict access to the results for up to five years.

The communication is done over a fully switched Ethernet backplane. The normal communication pattern is that the Supervisory Unit (SU) sends a request to one of the cards and gets a single response packet. A fully switched network offers full duplex communication, so that frame collisions are avoided [3]. Since the normal CSMA/CD protocol is no longer required a deterministic transfer time can be calculated based on the speed of the switch.

The network topology of the whole system is shown in Figure 2, and the topology of a single motherboard is shown in Figure 1. The maximum number of motherboards and cards is shown in these diagrams. The system can be configured with fewer motherboards, and also fewer cards per board.

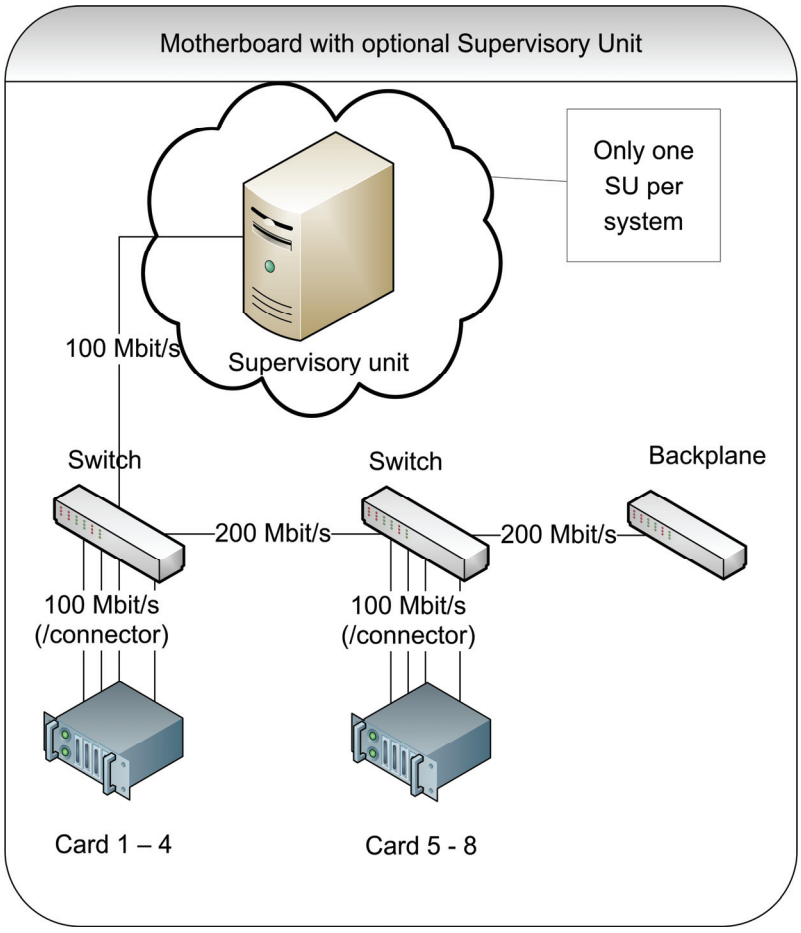

Figure 1: Target system motherboard layout

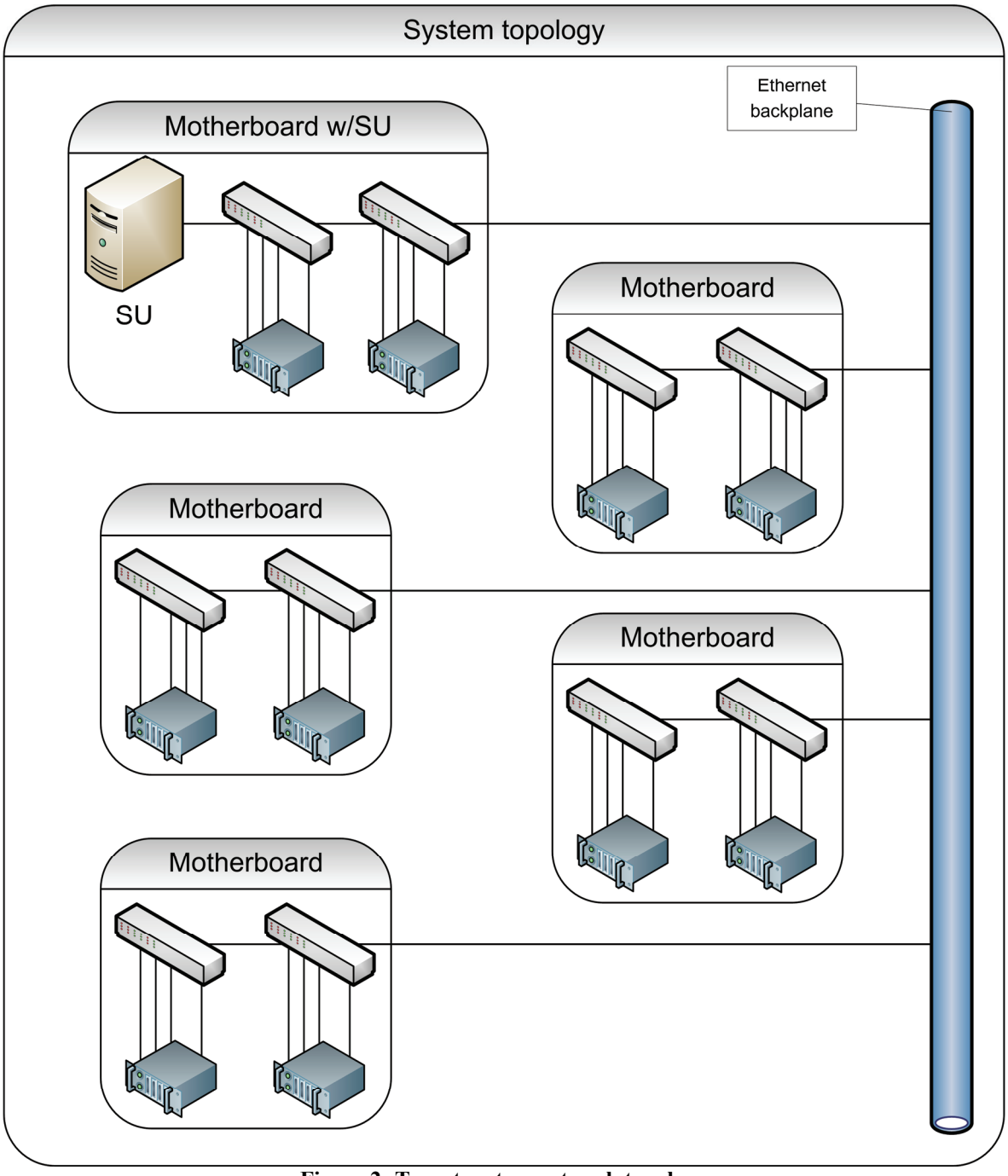

Figure 2: Target system network topology

1.3.1 The Problem: Transient periods of extended delays

Automatic boot configuration of cards takes up nearly all available bandwidth for extended periods of time. This will happen sporadically, and will severely delay alarms, diagnostics messages and similar types of low-volume traffic.

Nera wants to be able to achieve a deterministic upper limit on packet transmission times. This will serve to uphold the static priority scheme which is used in the RTOS scheduling, and will also ensure that the problems related to introducing duplicates upon retransmission can be completely avoided.

2 Project plan

I created the following project plan at the beginning of the semester based on a template from Nera. A description of alterations and an evaluation of this original project plan are given in chapter 11.

2.1 Timeline

Task																			
COTS Analysis																			
Problem definition																			
Requirement Specification																			
Test Plan																			
Design Documentation																			
Implementation								*											
Test Report																			
User Documentation																			
Project report								*											
Week number	4	5	6	7	8	9	10	11	12*	13	14	15	16	17	18	19	20	21	22	23

Table 1: Initial project timeline

(* Easter)

2.2 Deliveries

Requirement Specification
Form: Word document **Due date: 01.02.2007, week 5**

Test Plan
Form: Word document

Test Overview (before coding starts) **Due date: 22.02.2008, week 8**
Final Form with details (after coding completed) **Due date: 09.05.2008, week 19**

Design Documentation
Form: Word document/UML schematics

First version (before coding starts) **Due date: 07.03.2008, week 10**
Final version (after coding completed) **Due date: 18.04.2008, week 16**

Implementation
Form: .c/.h programming files

Final **Due date: 11.04.2008, week 15**

Test Report
Form: Word document **Due date: 09.05.2008, week 19**

User Documentation
Form: Word document **Due date: 09.05.2008, week 19**

3 User Documentation

3.1 Introduction

This chapter serves three purposes:

- To give a brief introduction to the design of the Priority Based Message Stack (PBMS)
- To explain the API of the stack
- To describe how a priority scheme is configured

PBMS is based on the open source project lightweight IP (lwIP). Whenever a feature of PBMS is essentially unaltered compared to lwIP, the lwIP documentation is also a good source for information. It is available from the Savannah project page [4] and the ScribbleWiki [5]. PBMS is based on the code from lwIP version 1.3.0-stable, including the bug fixes up until 2008-05-09.

LwIP is intended to be run as a standalone user space process, and its design reflects this. It has a single thread that manages all transmission, reception and memory handling. This thread will be referred to as the Tcpip thread. The user program runs in a separate thread which uses the BSD socket interface to PBMS / lwIP. An additional thread, or interrupt service routine, handles data reception in the network interface. This will be referred to as the Netif thread. The design is detailed further in the software architecture and design descriptions, see chapters 6 and 7.

LwIP has a clearly defined generic OS interface, and also a generic network interface. Both will be implemented specifically for the Integrity RTOS. The key features lwIP requires are threading, semaphores and message queues. There is an existing implementation of a message queue which only requires a generic semaphore construct, so the target OS does not have to offer message queues. PBMS has no additional platform requirements compared to lwIP, so the lwIP documentation is an adequate reference for the procedure of porting PBMS.

LwIP seems to be a widely used open source project. Although many of the links are now broken, Adam Dunkels' lwIP link site [6] indicates that the protocol stack has been used successfully by several companies, as well as research projects.

3.2 BSD Socket Interface to PBMS

The socket interface to lwIP is a limited implementation of the standard BSD interface [7]. As far as possible the PBMS socket interface has retained this similarity, although the syntax of return values from the ***sendto*** and ***recvfrom*** functions have been altered.

In the current source there are compile options that may be used to rename the standard socket functions. In the source files where they are declared and defined, all socket functions have the prefix ***lwip_***. This prefix will be used when discussing specific code segments. The option ***LWIP_COMPAT_SOCKETS*** can be used to enable the normal socket API function names. All options are configured in the file lwipopts.h.

3.2.1 Nonblocking IO

Nera required nonblocking operations, but ***lwip_sendto*** was only implemented in blocking mode. One potential blocking point within the stack is the queue of messages to the Tcpip thread. The PBMS implementation of ***sendto*** will return ***EWOULDBLOCK*** if that queue is full, instead of waiting for the message to eventually be posted.

In PBMS *sendto* and *recvfrom* return the value *–EWOULDBLOCK* in the event that the functions would have blocked. This is not in accordance with the standard socket interface.

In a normal BSD socket implementation ***sendto*** and ***recvfrom*** return -1 and set the socket error number to ***EWOULDBLOCK***. To determine that the error was in fact ***EWOULDBLOCK*** it is then necessary to use ***getsockopt***.

To avoid having to prioritize calls to ***getsockopt*** the actual error value is returned in the case of ***EWOULDBLOCK***, not just -1. The error condition is normally cleared by calling ***getsockopt***, but since we want to avoid calling that function ***EWOULDBLOCK*** is only used as the return value, and is not set for the socket.

Nonblocking receive was implemented in lwIP, and the syntax of return values was altered to be in line with ***sendto***.

The standard functions ***sendmsg*** and ***recvmsg*** are not implemented in lwIP or PBMS.

All calls to socket API functions not related to IO are handled at an equal (user-defined) priority, so intermixing calls to ***sendto*** with for instance ***getsockopt*** may invert the priority scheme. If such intermixing is desired, then the whole socket API should be priority differentiated based on the TOS stored in the calling protocol control block (PCB). This is a feasible modification to PBMS that would mainly be done in the Tcpip thread and the priorities header, which is described in chapter 3.2.4.

Such a modification may degrade the priority scheme, as each priority level must handle more packets from system calls unrelated to message transmission. The worst-case time for a transmission will therefore increase.

3.2.2 Enabling nonblocking receive – *ioctl*

Blocking mode is the default for a newly created lwIP socket. The nonblocking send is always enabled in PBMS; the option is not configurable as of yet. Putting a socket in nonblocking receive mode is done like this:

```
/* ioctl_arg: 1=nonblocking, 0=blocking */
ioctl_arg = 1;
lwip_ioctl(socket2,FIONBIO,&ioctl_arg);
```

Nonblocking receive can also be done using the ***MSG_DONTWAIT*** flag for each separate operation, as in the standard BSD interface.

3.2.3 Socket thread safety

The lwIP / PBMS BSD sockets are not thread safe. Specifically, two threads can not operate simultaneously on the same socket. A procedure where one thread does all initialization, and another does all transmission after init is complete is not a problem. For further information, see lwIP-users mailing list [8] (2008-05/msg00080.html and 2008-05/msg00082.html)

3.2.4 Priority assignment per socket

The priority for a newly created socket is by default 0, corresponding to the IP Type-Of-Service routine precedence. To give all packets sent from that socket higher priority one must use the ***setsockopt*** system call. The priority level must be chosen from the range of TOS precedence levels:

```
IPTOS_PREC_NETCONTROL           0xe0
IPTOS_PREC_INTERNETCONTROL      0xc0
IPTOS_PREC_CRITIC_ECP           0xa0
IPTOS_PREC_FLASHOVERRIDE        0x80
IPTOS_PREC_FLASH                0x60
IPTOS_PREC_IMMEDIATE            0x40
IPTOS_PREC_PRIORITY             0x20
IPTOS_PREC_ROUTINE              0x00
```

For example like this:

```
int s;
int set_tos;
socklen_t set_tos_optlen;

s = lwip_socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

set_tos = IPTOS_PREC_PRIORITY;
set_tos_optlen = sizeof(set_tos);

lwip_setsockopt(s, IPPROTO_IP, IP_TOS, &set_tos, set_tos_optlen );
```

The current socket priority level can be examined using the ***getsockopt*** function. Upon successful completion the value is stored in the variable ***tos_optval***.

```
int tos_optval;
socklen_t tos_optlen;

tos_optlen = sizeof(tos_optval);

lwip_getsockopt(s, IPPROTO_IP, IP_TOS, &tos_optval, &tos_optlen );
```

3.3 Configuring a priority scheme

The priority based message stack (PBMS) supports eight different priority classes, corresponding to the eight precedence levels of the IP type-of-service bits. Note that the stack support eight levels for input and eight levels for output, and that input and output priorities can be completely independent of each other.

All configuration of the priority scheme is done at compile time through the file *priorities.h*. The user can choose how many different priority levels that is to be supported. The stack is managed by the Tcpip thread, and the number of priority levels equal the number of different message queues this thread will service.

3.3.1 Number of priority levels

The first options to set are:

```
PBMS_PRIORITY_LEVELS       2
PBMS_DEFAULT_PRIORITY      0
```

The sample setting above will set up the system with two message queues into the Tcpip thread, one for normal priority traffic and one for high priority traffic. The scheduling is strictly priority based.

As long as there is a high priority message in one of the queues the low priority traffic must wait.

The range of assignable priorities will be ***[0, PBMS_PRIORITY_LEVELS-1]***, which in this case is only 0 or 1, where the highest numerical value has the highest priority. Using a labeled default priority in the configuration setup is optional.

3.3.2 Input priority

For each incoming IP packet the TOS value is checked, and for each of the eight TOS values there is a corresponding ***PBMS_INPUT*** priority. These are defined as such:

```
PBMS_INPUT_PREC_NETCONTROL         1
PBMS_INPUT_PREC_INTERNETCONTROL    1
PBMS_INPUT_PREC_CRITIC_ECP         1
PBMS_INPUT_PREC_FLASHOVERRIDE      1
PBMS_INPUT_PREC_FLASH              1
PBMS_INPUT_PREC_IMMEDIATE          1
PBMS_INPUT_PREC_PRIORITY           1
PBMS_INPUT_PREC_ROUTINE            0
```

This example scheme will give equal priority to everything above routine traffic. As long as there are no network elements that make use of the highest TOS values, such as ***NETCONTROL***, the user can set up an input priority scheme using as much as eight different levels.

3.3.3 Reception priority

Socket packet reception (***recvfrom***) is executed via the Tcpip thread. To maintain an unbroken chain of prioritized function calls this must be priority-enabled as well. Again it is up to the user to configure the mapping from socket TOS to message queue:

```
PBMS_RECV_PREC_NETCONTROL        1
PBMS_RECV_PREC_INTERNETCONTROL   1
PBMS_RECV_PREC_CRITIC_ECP        1
PBMS_RECV_PREC_FLASHOVERRIDE     1
PBMS_RECV_PREC_FLASH             1
PBMS_RECV_PREC_IMMEDIATE         1
PBMS_RECV_PREC_PRIORITY          1
PBMS_RECV_PREC_ROUTINE           PBMS_DEFAULT_PRIORITY
```

3.3.4 Output priority

The output priority scheme is quite similar to the input scheme. The TOS value for each outgoing packet is fetched from the PCB, and the packets are queued according to it. The macros that are to be defined are:

```
PBMS_OUTPUT_PREC_NETCONTROL        1
PBMS_OUTPUT_PREC_INTERNETCONTROL   1
PBMS_OUTPUT_PREC_CRITIC_ECP        1
PBMS_OUTPUT_PREC_FLASHOVERRIDE     1
PBMS_OUTPUT_PREC_FLASH             1
PBMS_OUTPUT_PREC_IMMEDIATE         1
PBMS_OUTPUT_PREC_PRIORITY          1
PBMS_OUTPUT_PREC_ROUTINE           0
```

It is important to keep in mind that the input and output priority levels can be assigned independently of each other. Look at this example scheme:

```
PBMS_PRIORITY_LEVELS        3

PBMS_INPUT_PREC_PRIORITY   2
PBMS_INPUT_PREC_ROUTINE    0

PBMS_RECV_PREC_PRIORITY    PBMS_INPUT_PREC_PRIORITY
PBMS_RECV_PREC_ROUTINE     PBMS_INPUT_PREC_ROUTINE

PBMS_OUTPUT_PREC_PRIORITY  1
PBMS_OUTPUT_PREC_ROUTINE   0
```

The Tcpip thread now services three message queues. Input and output traffic with routine priority is put into the first queue, with index 0. Packets with equal priority are processed in a FIFO manner. Priority output traffic will be handled before routine input and output, but priority input traffic will always be serviced first. A steady stream of input priority signals will result in complete starvation for all other types of traffic.

3.3.5 Internal stack management signals

In addition to incoming and outgoing packets, the Tcpip thread is responsible for handling internal operations that require mutually exclusive access to PBMS core functions, such as timers and memory handling.

It is possible that the internal timers can experience starvation, given that the stack prioritizes strictly. The only timer used by default if for ARP, and has a period of five seconds.

It is not advised to use the available PBMS timer facility; use a platform specific facility instead.

The priority scheme is primarily designed to service low-volume priority traffic in the presence of large low priority transmissions, so extended periods of starvation are not to be expected. The tests have so far not triggered timing errors due to large amounts of priority traffic.

The following options correspond to different socket operations:

```
PBMS_NEWCONN_PRIORITY      0
PBMS_DELCONN_PRIORITY      0
PBMS_BIND_PRIORITY         0
PBMS_CONNECT_PRIORITY      0
PBMS_DISCONNECT_PRIORITY   0
PBMS_GETADDR_PRIORITY      0
```

This option is used for memory handling and ***set*/*getsockopt***.

```
PBMS_CALLBACK_PRIORITY     0
```

This option is currently only used for processing the ARP timer, which has a period of 5 seconds.

```
PBMS_TIMEOUT_PRIORITY      0
```

3.4 Using other lwIP modules

Those lwIP modules that are not included in PBMS have not been tested in any way. The result of including them is generally unknown. Even if these modules compile without error there can be several unpredictable run time errors.

3.4.1 BOOTP and DHCP

The PBMS BOOTP server has not been tested in any way together with the lwIP DHCP client module. As they only use one of the reserved UDP ports each it is theoretically possible to have both a DHCP client (at port 68) and a BOOTP server (at port 67) running simultaneously, but this has not been a design goal in any way.

3.4.2 TCP

The lwIP TCP module has not been tested with PBMS. Altered features in PBMS that might affect TCP are for instance the Tcpip thread, the priority scheme and the nonblocking send feature.

3.5 Statistics and debug prints

3.5.1 Statistics

The built-in statistics are quite useful during test and debug. In the header stats.h the only externally visible part of the interface is defined: ***void stats_display(void)***

By setting the options ***LWIP_STATS*** and ***LWIP_STATS_DISPLAY*** in lwipopts.h ***stats_display*** can be used during normal program execution. Which stats that is to be displayed is also configurable in lwipopts.h.

3.5.2 Debug module

Debug printouts, which normally would be done using ***printf***, are treated in an OS generic manner, and can be enabled and disabled simple compile options. The general interface is:
LWIP_DEBUGF(flag,message);

The message can be formatted like a normal ***printf*** operation, given that the actual platform specific function can handle that.

Classes of debug messages are enabled and disabled in lwipopts.h.

4 Requirement Specification

4.1 Introduction

This document describes the requirements for a priority based network stack which is to be used in Nera's system. In accordance with the recommendations in IEEE 830-1998 [9] the requirements are either given as software requirements or process requirements.

The software requirements are divided into functional and non-functional requirements. The degree of necessity is stated for each requirement, with the following categories:

- **Essential:** Requirement must be fulfilled for product to be acceptable
- **Conditional:** The requirement would enhance the product, but absence does not make the product unacceptable.
- **Optional:** A class of functions that may or may not be worthwhile to implement.

4.2 Software Requirements Specification

This chapter details requirements that specifically focus on the software of the system. These are subdivided into functional requirements and quality requirements (interfaces, performance and modifiability).

4.2.1 Functional Requirements

Hierarchy	ID	Name	Type	Description
1	PBMS1000	Priority Based Message Stack	Collection	These are the requirements for a priority based message stack developed for Nera.
1.1	PBMS1100	Message scheduling	Functional, Essential	The message stack must enable strict prioritizing between different classes of messages.
1.1.1	PBMS1110	Non-preemptive message scheduling	Functional, Conditional	Whenever a high priority message is queued up, no message of lower priority should be sent out. In the event that a low priority message is being transmitted at the same time as a high priority request arrives, the low priority message is allowed to finish before high priority traffic commences.

4.2.2 Software interface requirements

Hierarchy	ID	Name	Type	Description
1.2	PBMS1200	Protocol support	Collection, Essential	The message stack must support the IPv4, BOOTP, ARP and UDP protocols.
1.2.1	PBMS1210	UDP	Functional	RFC 768 [10], but not full UDP: - no CRC - no fragmentation - no IOControl - only non-blocking function calls
1.2.2	PBMS1220	BOOTP	Functional	RFC 951 [11]– Bootstrap Protocol RFC 1542 [12] – Clarifications and extensions for BOOTP
1.2.3	PBMS1230	ARP	Functional	RFC 826 [13] – Ethernet ARP
1.2.4	PBMS1240	IPv4	Functional	RFC 791 [14] – Internet Protocol RFC 894 [15] – IP over Ethernet

Hierarchy	ID	Name	Type	Description
1.3	PBMS1300	Interface definition	Non-functional, Conditional	The communication layer interface should mimic the industry standard BSD sockets. Priorities can for instance be assigned through the ***setsockopt*** system call.

4.2.2.1 Framework of implementation

The priority based message stack will run as a standalone module accepting API requests through an RPC mechanism. The stack will dispatch messages to and receive messages from the physical medium through a device driver, which is referred to as ***IODevices*** in requirement PBMS1610. See the system outline in chapter 5.3.

The device driver interface and API RPC mechanisms will be specified and handled by NERA Networks and are to be considered peripheral to the main goals of this assignment.

Hierarchy	ID	Name	Type	Description
1.6	PBMS1600	Interface to existing modules	Non-functional	The messaging layer will accept API requests through an RPC mechanism and dispatch and receive messages toward the physical layer through a device driver.
1.6.1	PBMS1610	***IODevices*** can not be altered	Non-functional	The Ethernet device driver module ***IODevices*** can not be altered.

4.2.3 Performance requirements

Hierarchy	ID	Name	Type	Description
1.4	PBMS1400	Deterministic RTT	Non-functional, Essential	The main performance requirement for the priority based network stack is to offer a deterministic worst-case round-trip time, as shown in the figure below.
1.4.1	PBMS1410	Average throughput	Non-functional, Conditional	Average RTT should not increase more than 15 % compared to the current implementation. A value for average RTT in current system is to be determined at a later time.

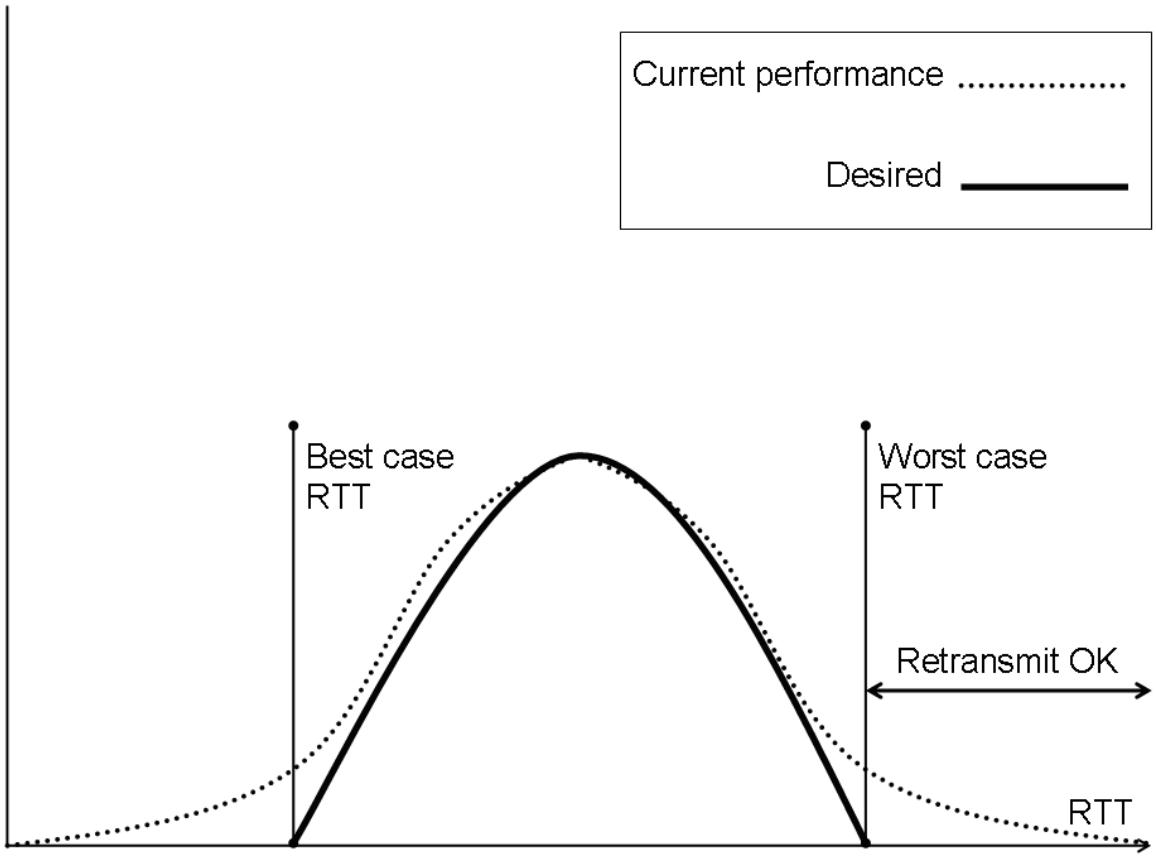

Figure 3: Performance requirement for PBMS

Hierarchy	ID	Name	Type	Description
1.5	PBMS1500	Memory footprint	Collection	The two following requirements describe the memory constraints for the implemented solution.
1.5.1	PBMS1510	Size of management module	Non-functional, Conditional	The management module (static memory) part of the protocol stack can be at most 75 Kb when compiled for an ARM processor or similar computer architecture.
1.5.2	PBMS1520	Size of message queues	Non-functional, Conditional	The amount of memory allocated for message buffers is limited by the efficiency of the chosen implementation. Only a few percent of the allocated memory should be used for management structures.

4.2.4 Other quality requirements

4.2.4.1 Modifiability

- It should be possible to alter the scheduling policy of the priority based message stack within a reasonable amount of time. This requirement can be disregarded if it causes problems in the modularization and implementation of the system.

- Potential changes in network structure that should be taken into consideration:
  - Adding more motherboards to the system

- Changes in network structure that will NOT take place in the foreseeable future:
  - The addition of routers or other elements that may use IPv4 TOS coding for internal messaging
  - Replacing the switches with a kind that does not support forwarding based on IPv4 TOS value.

4.3 Process requirements

These requirements describe how PBMS must be implemented in order to function with the target system.

Hierarchy	ID	Name	Type	Description
1.7	PBMS1700	Implementation language	Non-functional	The implementation should be done in standard ANSI C (ANSI X3.159-1989 "Programming Language C.").
1.7.1	PBMS1710	Kernel space language	Non-functional	All Integrity kernel space modules must be implemented in ANSI C.
1.7.2	PBMS1720	Stack space language	Non-functional	Stack space modules can be implemented in either C or C++.

Hierarchy	ID	Name	Type	Description
1.8	PBMS1800	COTS requirements	Non-functional	Any open source SW or other kinds of COTS that are to be used in the implementation must have a BSD-type license.

5 Component-Based Software Development

5.1 Introduction

This chapter begins with a general introduction to a software development process which is based on the use of off-the-shelf components (COTS). A description of the available technologies and components suitable for this specific project are then presented in chapter 5.3. One of the described technologies, a per-packet priority scheme, is chosen. Several software components that implement this technology are then evaluated. Chapter 5.5 finally describes the process of analyzing the chosen component; lwIP.

5.2 The applied software development process

The project plan in chapter 2 might give the impression of a waterfall [16] development process. The COTS aspect of this project does however make it quite different from that traditional process. In the following subchapter a workflow for software engineering using COTS is described.

5.2.1 Component-Based Design

This development method is described in [1]. In a software development process these steps can be carried out sequentially, as in iterative development. If the process ends at 6b or 6c, a new model solution is searched for, found and tested. The steps can also be done in parallel, where one would evaluate several model solutions simultaneously. Large-scale parallel model evaluation can be quite costly [1], so it is advised to have one main model and perhaps a few minor evaluation efforts on the side.

1. A **design question** is identified.
   → For this project the design question is the initial problem definition, as presented in the beginning of this report. To be in line with the terminology of the source [1], the problem could be reformulated as "*Can this proposed solution offer deterministic transfer times for UDP packets?*"

2. **Starting evaluation criteria** must be determined. This describes how a potential solution is to be verified according to the *design question*.
   → In this project this is the test plan, given in chapter 8. The single most important test is to measure a deterministic transfer time.

3. **Implementation constraints** specify any part of the implementation context that will affect the implemented solution.
   → This is essentially the requirements specification given in chapter 4, which for instance gives details about implementation language, required licenses for any OSS, required protocol support, and many other implementation aspects.

4. A **Model solution** is a minimal COTS-based implementation that potentially satisfies the *starting evaluation criteria* and *implementation constraints*. An important point here is to study the component features that are most likely to support or contradict the *design question*.
   → For this project this would be the end result implementation, PBMS. This is described in the architecture document (chapter 6).

5. A set of **ending evaluation criteria** is created to capture experiences gathered while implementing the model solution, and are used together with the *starting evaluation criteria*.
   → Implementation experiences from this project have been captured and documented in the design and architecture documents, test plan and test report (chapters 6 - 9). The tests must for instance ensure that the modifications listed below actually work, and that they do not introduce errors elsewhere in the stack.
   a. Two bugs related to full message queues that were found in lwIP and attempted fixed in PBMS.
   b. The addition of nonblocking send in PBMS.

6. **Model evaluation**. The proposed model solution is evaluated according to the criteria. Possible outcomes are:
   a. An accepted model solution
   b. Rejection of model solution
   c. New or altered design questions

5.2.2 Adaptations of the development process

The *model evaluation* of PBMS is primarily to carry out the tests given in the test overview, and evaluating the results. A general discussion and conclusion is then given in chapters 11 and 12.

For large systems a model solution will often consist of an ensemble of several components. In this project most of the work has been related to a single software component (lwIP), but the test platform (Linux) and target RTOS (Integrity) combined with lwIP could be seen as the total ensemble of software components.

LwIP is strictly not an Off-the-shelf component in this project, since several key functions of it have been altered to make PBMS. If lwIP is viewed as a collection of several modules, then most of them are in fact used unmodified. Most of the changes have been done in the Tcpip thread sockets interface.

5.3 Nera's design considerations for PBMS

This subchapter details several design considerations Nera gave concerning the implementation of PBMS. These were used in the COTS selection process, which is described in chapter 5.4.

The following sketches give an overview of a part of the existing software architecture as well as a desired architecture for the Priority Based Message Stack (PBMS). The stack will be used in the Supervisory Unit (SU), see the system topology diagrams in chapter 1.3.

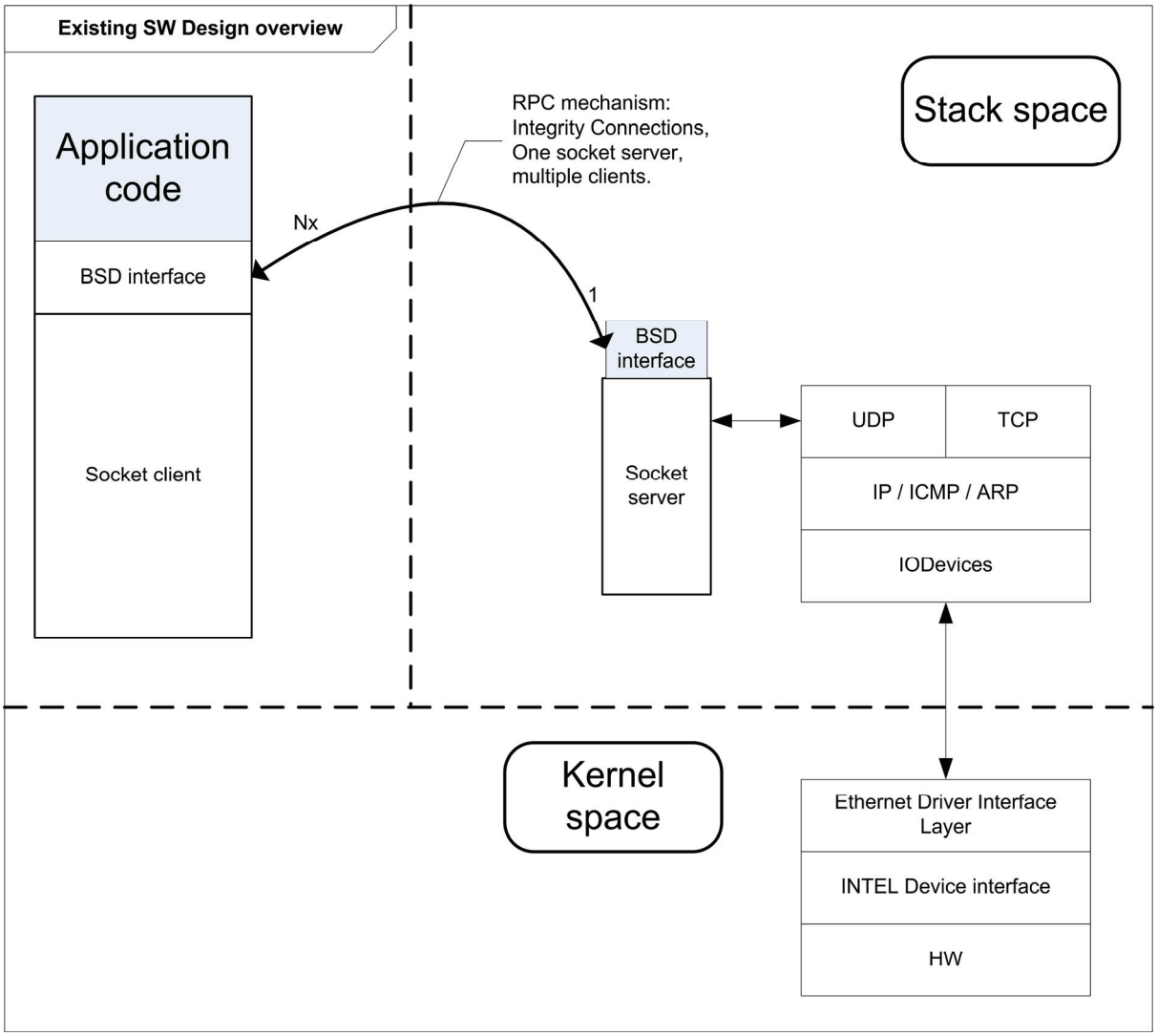

Figure 4: Existing software design

In the following diagram the question of placing the stack in kernel space depends on if it is implemented entirely in ANSI C, and a general evaluation of the possibility that the stack will introduce errors.

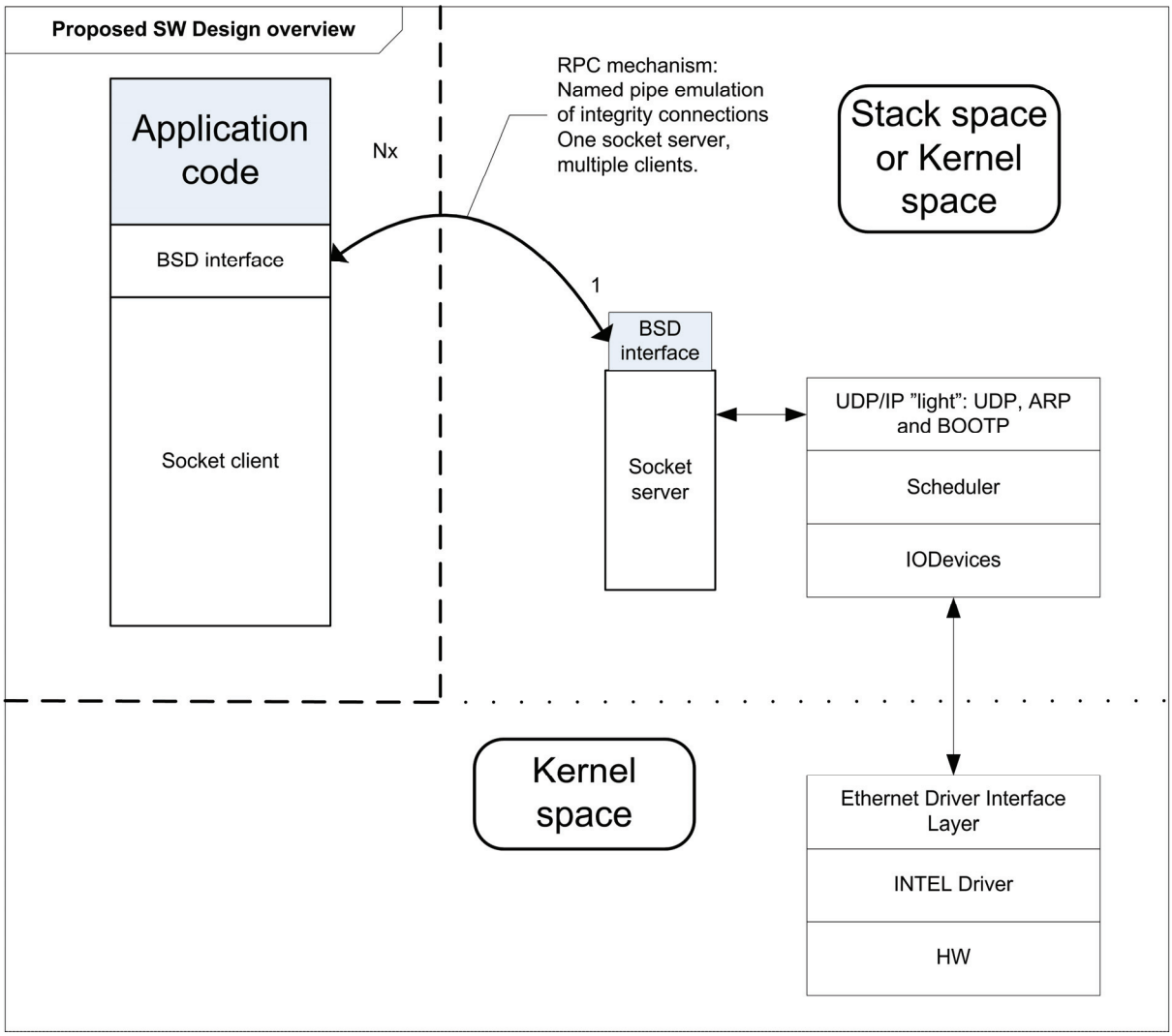

Figure 5: Proposed software design

5.3.1 BSD Socket interface

The applications that will run on top of PBMS only use nonblocking socket calls. When it comes to the BSD socket ***select*** function, keeping track of exception state is not necessary for connectionless sockets. Mapping socket to port, and port to socket, should preferably be done in constant time.

Using ***setsockopt*** the priority of a socket can be raised or lowered at any time. Moving messages from one queue to another because of altered socket priority is not a desired feature.

5.3.2 Memory handling and partitioning

PBMS should have a compile time option to configure the number of message buffers. The buffers can be of a single fixed size or come in different sizes. The existing stack offers the following sizes; small: 64 bytes, 128 bytes, medium: 512 bytes. Many packets will be only 64 bytes.

Do memory allocation only at start-up. Let the efficiency (percentage of memory devoted to management structures) limit the amount of memory allocated to message buffers.

The buffering scheme must also be considered in relation with the priority assignment. There are two general options:

- One pool of buffers per priority level
- A common buffer pool with the possibility of freeing/dropping low priority messages when needed.

5.3.3 Message reception buffering

Message receive buffers may fill up. There are two options:

- Keep messages in queue, discard newly arrived messages.
- Discard the oldest low priority messages

The alternative is flow control, which would be more complex. Full queues are a symptom of an error somewhere else in the system (higher up). It may be sufficient to add a hook for logging of such error conditions.

When priorities are taken into consideration the following scheme is an option:
Event: Incoming packet P1.

- If queues are full and P1 is in the lowest priority class, drop P1.
- If queues are full and P1 is not in the lowest priority class, drop a package in the lowest priority class and keep P1.

The number of receive queues (priorities) should be a compile time parameter. Include the possibility of defining a maximum size on receive queues.

5.3.4 Modularization

To make the implementation portable between a Linux test platform and the target OS an OS abstraction layer can be implemented.

5.3.5 TOS / DSCP

It is preferable to choose the TOS / DSCP values freely from the whole range, and then map them down to four levels, as the switches only have four queues. There will then have to be a compile time option for the number of available priority levels.

5.4 PBMS COTS Analysis

This subchapter describes the search for suitable software components to use in the development of PBMS. A general evaluation of this process is given in chapter 11.3, but most of the selection process is presented here. The next subchapter is a detailed description of how the chosen software component, lwIP, was analyzed.

According to the development process described in chapter 5.2.1 a necessary prerequisite to undertake a search for components is to have the problem definition and requirements specification at hand. As can be seen from the planned project timeline in chapter 2.1 the initial idea was to do the search at the same time as the requirements were being documented.

The two following subchapters detail how we[1] first investigated the different technologies that could be applied. Afterwards I focused on finding the most suitable software components that implemented the TOS precedence or DSCP standard, described below.

[1] Trygve Lunheim took part in the technology analysis described in this chapter.

5.4.1 Candidate technologies

We have in this project investigated technologies at two different levels of the OSI model [17], the network layer and the data link layer. Given the required protocol support (UDP/IP) several layer 4 protocols, such as SCTP and RSVP, were only briefly investigated, as they did not seem to be viable alternatives. The comment from Nera below concerning no additional capacity on some microcontrollers is also relevant for the use of RSVP or SCTP. Adding support for RSVP or a similar scheme would require additional code in all network elements.

As Nera had no intention to alter the data link layer in the system, for instance by using a CAN bus or a token ring network, such options have not been considered either. This led us to evaluate any layer 2 or layer 3 protocols operating on standard Ethernet.

5.4.1.1 Layer 2 - Priority based Ethernet

An implementation of the standards IEEE 802.1Q, IEEE 802.1p and IEEE 802.3ac [18] makes it possible to offer strict prioritization between different packets at layer 2 of the OSI model. The basic principle is to extend the maximum size of the Ethernet frame with 4 bytes, and use these bytes to denote the priority level.

This option was evaluated early on, and Nera gave the following response:

Generally it is desired to have backwards compatibility with all other network elements in the new protocol stack. Some of the cards that are used in the network have very simple microcontrollers, where the code size has been pushed to the limit. As it is now those would simply not have storage capacity for the additional code to handle the extra tag in 802.3ac.

This option was therefore not considered any further.

5.4.1.2 Layer 3 – Quality of Service mechanisms

IP-based layer 3 QoS mechanisms do generally come in two categories [19]:

- Packet classification
- Resource reservation

The latest version of the 'per packet' mechanism is known as DiffServ [20], and is discussed in the next paragraph. Integrated Services as presented in RFC 1633 can be implemented by using the RSVP protocol. As mentioned previously that protocol was not suitable for this specific system. The rest of this subchapter is therefore a discussion of the packet classification approach to QoS.

Any IPv4 per-packet priority scheme is basically an interpretation of 8 bits of the IPv4 header, which in RFC791 [14] are referred to as the Type of Service (TOS) bits. The interpretation of the TOS field was updated in the now obsolete RFC1349 [21]. The proposed standard RFC2474 [20] gives the most recent interpretation of these bits, and the octet is now referred to as the DS field. The standards do not issue a normative description of how packets are to be forwarded based on this field; that is left to the implementers.

Nera were positive to a solution based on this technology:

> *On the microcontrollers mentioned above, using DiffServ seems to be a lot easier. An outgoing package (which is always sent as a reply to an incoming one) will just copy the DSCP field from the incoming packet. This way the priority is equal on both request and reply.*

The result of the general technology search was to implement a priority scheme based on the TOS octet of the IPv4 header.

5.4.2 The software component selection process

The primary capability of a suitable software component must be to generate IP packets with proper TOS or DSCP labels. Another key feature must be to give strict prioritization to certain packets streams within the network stack. The proper prioritization should be ensured once the packet reaches the network, but variable queuing delays can still occur inside the stack if all outgoing traffic is buffered together. These features, combined with implementation language, memory constraints and open source licenses were the most important evaluation points when searching for software components.

Since the Differentiated Services definition of the TOS octet is the latest proposed standard, the search for software components was initially centered on finding DiffServ implementations. I did for instance search the DiffServ-implementations mailing list for any descriptions of implemented solutions and related experiences. Discussions concerning DiffServ are most of the time at a higher level than what is relevant to this project. Major research issues are how to define priority schemes that will be valid across the entire Internet. When it comes to a single, isolated network like the one in this project the solution is actually quite simple.

The switches in the target system's network can be set up to forward packets strictly based on the TOS / DSCP field. See Figure 1 and Figure 2 in chapter 1.3. Assuming that all network elements of the target system work according to their specifications, the networking part of the project is already finished merely by choice of technology. The test procedures for verifying this claim are presented in chapter 8.

All communication in the target network is done in a request-response manner initiated by the Supervisory Unit. All response traffic only has to mirror the priority of the incoming request to uphold the priority scheme. There are no network elements (such as routers) that generate traffic using any of the TOS precedence classes, and it is explicitly stated in the requirements specification that no such elements will be added.

As mentioned I initially focused on DiffServ implementations. While these certainly can generate IP packets with the DSCP field set, they also offer a host of other features that are defined in the Differentiated Services architecture. The desired scheduling policy for PBMS is only strict prioritization. Most QoS schemes are much more elaborate, detailing Per Hop Behaviors and more complex scheduling algorithms [19].

Two General purpose TCP/IP Stacks

There are two easily available open source implementations of DiffServ; Linux with Traffic Control (TC) package or FreeBSD OS with ALTQ [19]. An implementation based on such a desktop OS network stack would then have to start with porting the stack to the target OS. This has already been done at least once with the Linux TCP/IP stack; LyraNET [22] is a zero-copy implementation of the Linux stack for embedded systems. LyraNET was written for LyraOS, an experimental operating system. Whether LyraNET supports the TC package is not known. Due to the license requirement (see PBMS1800 in chapter 4.3) anything Linux-derived was out of the question due to the GPL license. The alternative would then be to do a similar port of the FreeBSD TCP/IP stack. That seemed to be infeasible given the time constraints of this project.

A general purpose stack would probably not be in accordance with the required memory footprint, and require large modifications, as it depends on a certain OS interface. Based on this I shifted my attention to embedded TCP/IP stacks, which are more likely to fulfill more of the given quality requirements.

Embedded TCP/IP stacks
Two small TCP/IP stacks that are available under a BSD license are lightweight IP (lwIP) and micro IP (uIP) [23]. LwIP was chosen to be the basis for PBMS, and is described in the user documentation (chapter 3), chapter 5.5 and in the design document (chapter 6).

LwIP was chosen over uIP since one of the explicit design goals of lwIP is modifiability. According to [23] it is designed so that adding new protocol support should be easy. The uIP stack is targeted towards 8-bit systems, and does for instance use protothreads [24] to be suitable for such minimal systems. Protothreads are described as extremely lightweight stack-less threads designed for severely memory constrained systems.

Dunkels also reported that in the implementations of lwIP and uIP he found an inverse relation between memory footprint and performance [25]. Furthermore he recommended that designers who require a certain amount of throughput should choose the lwIP stack.

Considering that the target OS offers threading, that the stack does not have to run on 8-bit architectures, and that a certain throughput is required in the target system (for instance for boot configurations), lwIP was chosen over uIP.

Commercial stacks
There are quite a few commercial embedded TCP/IP stacks on the market. Two stacks that offered the desired protocols were Fusion[26] and NicheStack[27]. I sent queries to both concerning how the stacks used the TOS or DSCP field in the IPv4 header, but did not get any response. Since it is not known if these stacks could do the internal package prioritization that is desired, these were not considered any further.

Middleware - TAO
The ACE ORB (TAO) is described as a viable means for large-scale distributed real time and embedded (DRE) systems to achieve QoS in network communication [28]. TAO is implemented in C++ [29], and it is questionable whether it can fit in with Nera's memory footprint constraints [30]. The numbers referred to are measured for a regular desktop compile with GCC, so they can not be directly compared with the required binary size for the ARM architecture.

Even though Integrity RTOS offers third party integration with TAO [31], it seems to be out of the question for PBMS due to size constraints and implementation language. It is also not known whether TAO has the required performance characteristics and SW interfaces. In addition to this, Nera was familiar with this solution and did not want it to be used.

5.5 Working with lwIP

This subchapter describes the process of examining and modifying lwIP. The design and implementation of PBMS is described in chapter 6.

5.5.1 LwIP code lines

To give an impression of the work that has been done in this project I will start with some simple figures provided at the lwIP developers mailing list [32] (2008-04/msg00059.html). The numbers given are on the form code lines / total lines. The definition of a "code line" was not explicitly stated in the cited document, but some sample counts indicate that these are physical line of code (LOC) counts. The C preprocessor is used extensively in lwIP, so a simple semicolon count would be inaccurate.

The standard lwIP distribution, without any ports to specific platforms, consists of five text files and 119 source files, mostly organized as source and header pairs. LwIP is not functional in this form, it depends on a platform port to operate.

Module(s)	Code lines	Total lines
All of lwIP	34267	54864
lwIP files used in PBMS	18297	31232
lwIP files not used in PBMS	15970	23632
Code where the most modifications have been done	3582	5392

Table 2: LOC count for various parts of lwIP

5.5.2 Methods to evaluate the lwIP design

When evaluating whether an implementation of PBMS based on lwIP was feasible, there were not that many metrics available. Several sources claimed that lwIP was well modularized, and after having read through a lot of the code and done some modifications I tend to agree. But there was a lack of design documents that actually presented the design. Metrics such as functional cohesion, degree of coupling between modules, and the "uses" relation (described in the next subchapter), would have been very useful to know.

When facing a large collection of undocumented code there are certain tools that may be used to undertake an architectural reconstruction. The software architecture of the Linux kernel was reconstructed using the tools ***cfx***, ***grok*** and ***lsedit*** [33]. At that time the code base for the kernel was around 800 000 lines of code. The tools mentioned are currently available in the collection ***swagkit*** [34].

I decided not to follow an architecture reconstruction approach for several reasons.

1. The architecture is already known (supposedly)
2. The reconstruction approach seemed to be too complex for a project this size
3. I would have to spend an unknown amount of time on learning a specific toolkit
4. The result of the process could be to simply drop lwIP

The code size of lwIP is less than 5% of the quoted size of the Linux kernel, so the tool-based approach to code inspection might simply be too heavy in this project. I was also unfamiliar with the toolkit, and learning to use it properly could take too much time compared to the potential benefits.

The architecture of the protocol stack is supposed to be the well known layered architecture, but verifying that claim is not an explicit goal in this project. If the important parts of the lwIP stack were indeed poorly modularized, it should be possible to observe that early on from relatively simple inspections. Poor modularization would indicate that lwIP should not be used. Then it would not be a good idea to spend much of the allotted time on applying the reconstruction tool only to dismiss the component and start something else from scratch.

It is a general advice in [1] to use a least-effort method for architectural reconstruction, and in this case the least effort seemed to be manual analysis. An architectural overview can ultimately only give an indication of the modifiability. It seemed to me that the best strategy would be to just try and do the modification, and evaluate the effort afterwards.

5.5.3 The “uses” relation compared to the C “include” relation

This subchapter briefly discusses why it is difficult to reason about the modularization of a system based on the *include* relations.

File ”lwip/udp.h”:	**File ”lwip/udp.c”:**
#include "lwip/opt.h" #include "lwipopts.h" #include "lwip/debug.h" #include "lwip/arch.h" **#include "lwip/pbuf.h"** #include "lwip/opt.h" #include "lwip/err.h" **#include "lwip/netif.h"** #include "lwip/opt.h" #include "lwip/err.h" #include "lwip/ip_addr.h" #include "lwip/pbuf.h" #include "lwip/inet.h" **#include "lwip/ip_addr.h"** #include "lwip/opt.h" #include "arch/bpstruct.h" #include "arch/epstruct.h" **#include "lwip/ip.h"** #include "lwip/opt.h" #include "lwip/def.h" #include "lwip/arch.h" #include "lwip/pbuf.h" #include "lwip/ip_addr.h" #include "lwip/err.h" #include "arch/bpstruct.h" #include "arch/epstruct.h" **#include "arch/bpstruct.h"** **#include "arch/epstruct.h"**	**#include "lwip/udp.h"** **#include "lwip/def.h"** **#include "lwip/memp.h"** **#include "lwip/inet.h"** **#include "lwip/inet_chksum.h"** **#include "lwip/ip_addr.h"** **#include "lwip/netif.h"** **#include "lwip/icmp.h"** **#include "lwip/stats.h"** **#include "lwip/snmp.h"** **#include "arch/perf.h"** **#include "lwip/dhcp.h"**

Table 3: Included headers in udp.h and udp.c

The left field of Table 3 shows the header files directly and indirectly included in the file udp.h, and on the right are only the direct *includes* in udp.c. All files listed above have additional *includes* within as well, so a complete *include*-graph would become quite large. These two files are the best approximation of a UDP module in the lwIP protocol stack.

The UDP module has access to the functionality of the IP layer, which is to be expected according to both the TCP/IP reference model [35], [36] and the OSI reference model [17]. The UDP module also has access to the functionality in the network interface, which is the layer below IP. This is maybe one of the layer violations Dunkels refers to in [23]. The UDP module also has access to several interfaces in the layer above; ICMP, DHCP and SNMP are all at layer 5 in the TCP/IP reference model.

The reference models for the layered architecture do not specify how exactly entities at different layers are to interact. The fact that the UDP module can use functions from several layer 5 modules is not necessarily an architectural inconsistency. This might indicate that the lwIP UDP module depends on the correct behavior in some layer 5 protocols. Then the system can be difficult to analyze for dependability, as there could be a circular dependency graph.

Even with a complete *include*-graph at hand, showing all indirectly included files, it is hard to determine which modules the lwIP UDP implementation really depends on for working properly. The direct *includes* are an indication, but one really needs to examine which methods from the different headers that are actually used. The *include* mechanism allows for using functions and variables from indirect *includes*, so to be certain these would have to be studied as well.

Parnas described the *uses* relation as ”*A uses B if there exist situations in which the correct functioning of A depends upon the availability of a correct implementation of B.*” [37] Based on previous knowledge of TCPIP stacks, one could assume that UDP *uses* IP, but one has to examine the implementations closely to verify this. Whether lwIP UDP *uses* all the modules directly or indirectly included in code has to be resolved by manual code inspection, or by an architecture reconstruction workbench.

If the *uses* relation had been properly documented for lwIP, then some of the work done in this project could have been simplified, for instance documentation, testing, modification and also simply understanding how the software operates. As it is, one has to carefully review all function calls and accessed variables to investigate module dependencies and interaction.
Including the header of a certain module is a necessary prerequisite for *using* a module as defined by Parnas. However, an included header file does not imply that the module is in fact *used*.

5.5.4 Available lwIP documentation

The lwIP project is mainly documented on a ScribbleWiki page [5]. Patches, bugs, active development efforts and the project mailing lists are available through an lwIP Savannah project page [4]. These sites and mailing lists are the best sources of up to date information.
At the Savannah site the following recommendation can be found:

Reading Adam's papers, the files in docs/, browsing the source code documentation and browsing the mailing list archives is a good way to become familiar with the design of lwIP.

There is an up to date Doxygen [38] lwIP code reference available from the lwIP ScribbleWiki. This is only a compilation of code comments and other descriptions which are already available at the lwIP ScribbleWiki.

Adam Dunkels, the creator of lwIP and uIP, has a number of publications related to lwIP that are now treated as historical documents. A link to his homepage can be found at the wiki. A final piece of documentation located at the wiki is a collection of architectural Rx flow diagrams. These are mostly related to TCP, but one details the flow from link input up until udp_input.

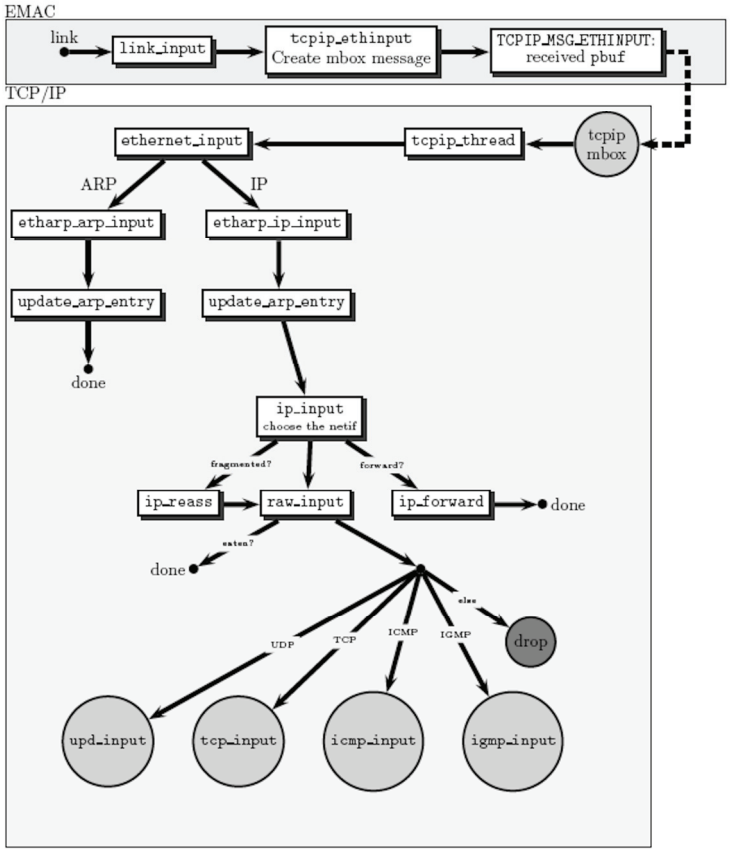

Figure 6: lwIP architectural Rx flow documentation

The diagram is not made according to any well-known standard, and it is initially not clear how it shows such things as:

- Thread behavior (dotted line)
- OS dependence
- Files referred to (not specified)

The UDP input was not detailed any further in the following diagrams, and a corresponding transmission diagram was not provided. I remade parts of this diagram in UML to understand and document PBMS. These can be found in chapter 6.

A description of the mapping between architectural elements (protocol layers) and source files was not available. For most layers this could be estimated by looking at folders and file names, but could only be verified through code inspection. According to Dunkels [23] layer violations are done to enhance performance in lwIP. This increases the cost of modifying the interfaces and/or replacing existing modules, especially the lower ones, as the entire stack might be using a given interface.

The way ARP packets are handled in this diagram is the normative procedure according to several discussions on the mailing lists, and according to version 1.3.0 of the lwIP source. However, several network interfaces in the Linux port would not send the entire Ethernet frames to the Tcpip thread for processing, leading to some confusion and problems with testing. This is detailed in the design document, see chapter 6.

6 The PBMS Software Architecture

6.1 Introduction

This chapter is laid out to be in conformance with the standard IEEE 1471-2000 [39] *Recommended practice for architectural description of software-intensive systems.* If and when PBMS is deployed in Nera's system the stakeholders can be described more accurately. As it is, my knowledge of the acquiring organization's internal structure is limited.

Some of the required elements are common for the whole report, and not stated specifically within this chapter:

- Date of issue and status
- Issuing organization
- Change history (this is in effect the first version)
- References

Scope and context
The scope of this document is currently twofold, it is part of my master's thesis, but is also intended to be the architectural description document for PBMS, which is implemented for Nera Networks AS. This will change when the thesis is handed in, and the stack is no longer a student project.

Glossary
The definitions of all software architecture terms can be found in [2].

6.2 Summary

This chapter describes the design and implementation of the Priority Based Message Stack (PBMS). Several features were already implemented in lwIP; in those cases this chapter documents how they fulfill the requirements. A basic introduction to the lwIP design is already given in the user documentation, see chapter 3.

The design rationale for PBMS is elaborated in the first subchapter. The implemented solution is to give strict priority to certain types of traffic, so that these should not be disturbed by high-volume, low priority communication. As described in chapter 3 the priority is assigned at the socket level.

6.3 The architecture of lwIP and PBMS

The reference architecture of lwIP is the TCP/IP model, as described in RFC1122 [35] and RFC1123 [36]. Since this architecture is well known and well documented elsewhere, only the known inconsistencies will be described here. The documentation of the existing lwIP architecture is described in chapter 5.5.4.

6.3.1 Known architectural inconsistencies in lwIP and PBMS

According to Adam Dunkels, the original creator of lwIP, layer violations are done to enhance performance in lwIP [23]. This increases the cost of modifying the interfaces and/or replacing existing modules, especially the lower ones, as the entire stack might be using a given interface.

PBMS does not use the ICMP module of lwIP, which is inconsistent with the requirements for internet hosts in RFC1122 [36]. This is not a concern as long as PBMS is not used as an internet host.

6.4 Design rationale

6.4.1 Strict priority queuing to ensure low delay

To ensure low delay over a network one possible strategy is to use a per-packet priority scheme [28]. The alternative is resource reservation, which was not found to be suitable in this project. A reservation scheme is more complex to implement, and PBMS is designed to handle transient periods of high load. Reserved bandwidth could be wasted most of the time. Per-packet priority schemes are also considered to be more scalable than bandwidth reservation schemes [40].

While many implementations use the DSCP or TOS field as a basis for a weighted priority scheme [28], [19], it is advised to implement strict priority queuing when dealing with delay sensitive traffic. Kos et al. [41] simulated priority queuing and found that it was suitable for delay sensitive traffic, while not degrading the overall performance of normal traffic significantly.

Schmitt and Zdarsky [42] argue that a strict priority scheme is simple, effective and available. The simplicity is multi-faceted:

1. Easy to implement in network elements (e.g. switches)
2. Easy to understand for users
3. Easy to analyze
4. Easy to configure and manage

The authors of [43] performed an in-depth measurement-based analysis of the Priority Queuing algorithm and found that it was one of the most effective algorithms to minimize queuing delay. One of their conclusions was that in the presence of stringent delay and jitter requirements the use of traffic aggregation (i.e. grouping several types of traffic in one priority class) must be limited. Their study also showed that the end-to-end delay is very sensitive to a first-come-first-serve (FCFS) policy in any network element, for instance the transmitting node.

In part due to the results in [43] PBMS prioritizes strictly all the way from socket interface to the network. Once a message is transmitted to the network the proper configuration of the switches (see diagrams in chapter 1.3) will ensure strict priority queuing all the way to the destination.

The current situation is that the switches can handle at most four classes of priorities in a strict scheme. The chosen solution leaves it up to the system architect or maintainer to decide how socket priority should be mapped to packet priority. In the implemented solution this scheme must be determined at compile time. This was the best minimum-effort modification of lwIP, but it is possible to implement a dynamically configurable scheme as well.

A protocol for the relation between OS level task priority and PBMS packet priority must be implemented manually. There is no currently no possibility of an automatic mapping between these attributes. This enables misuse of the system in the sense that someone might find out that a specific bulk transfer requires prioritization. The system is not designed for something like that, as high volumes of traffic at elevated priority can cause starvation both internally in the stack and in the network.

6.4.2 Possible extension: Run-time configuration

In a run-time configurable scheme there could be two available options:

1. The number of priority queues for the Tcpip thread
2. The mapping from TOS levels to Tcpip priority queue

A dynamic mapping from TOS to queue can be enabled with a minor modification of the current source, while a dynamic number of queues would require the use of another data structure. As the ANSI C language has no dynamic built-in data structure, this would have to be some kind of linked list. A method for receiving the updated configurations at run-time would have to be implemented. Receiving them in the form of a UDP packet is one option.

In a system which makes active use of static OS-level task priorities it could be interesting to be able to assign packet priority directly based on task priority. With the current switches this is clearly not useful, as a static scheme normally requires much more than the four levels offered by the switches, or 8 different TOS precedence levels. The DiffServ Code Point allows for 64 levels, which could be enough in a relatively small system. This would require some sort of query to the OS during the execution of the ***socket***() function, which then would initialize the DSCP of the socket.

6.5 Stakeholders and concerns

Disregarding the fact that this document is currently part of a thesis, the stakeholders are:
Nera Networks AS: Acquirers, potential maintainers, potential users.
Steinar Lieng Fredriksen: Developer.

The purpose of the system is defined within the problem statement and requirement specification of this report, the appropriateness is evaluated in the discussion and concluding chapters. The final evaluation of the software will be done after a test on the embedded target has been performed.

A risk related to the COTS based development is that one has little control over the quality attributes of the components. This can be addressed by rebuilding the stack from smaller separate components, as described in the following subchapter. This risk is currently an issue for all stakeholders.

6.5.1 Using lwIP as a set of components

It has been an explicit goal to use as much as possible of lwIP without modification, to quickly determine if the quality aspects of these components satisfy the given requirements.

As it is PBMS uses only a limited part of lwIP, but is compiled from the full source. One can perhaps achieve enhanced testability and modifiability by building a stack of lwIP components, but without many of the conditional includes. Some unused lwIP modules intermixed with the PBMS code are TCP, IP fragmentation, IP reassembly, Auto IP, checksums, and UDP Lite. Removing those would decrease the number of code lines in key control modules such as the Tcpip thread and socket API.

By building PBMS from smaller modules it would also be possible to document the “uses” relation (see chapter 5.5.3), which could enable dependability analysis, error containment, built-in testing, and lead to generally better understanding of the whole stack. Coding unit tests (for instance using the check framework [44]) is also an option. There is a current effort to code such tests for the entire lwIP stack, see task #7930 at [4]. Many unit tests designed for lwIP can be used directly with PBMS as well.

6.5.2 Lack of socket thread safety

It is mentioned in the user documentation that the lwIP sockets are not thread safe, see chapter 3.2.3. Considering the usage pattern outlined in *Figure 5: Proposed software design*, this can be considered a general architectural risk. There is a separate page[2] describing this in some detail at the lwIP ScribbleWiki [5]. This is one part of lwIP that seems to be work in progress, and might remain so for a while.

There are two possible solutions:

1. Solve this in the interface which processes the remote procedure calls from the user space applications. Operations on the same socket will have to be serialized. A thread that intercepts all remote procedure calls and manages the socket API is one option.
2. Make the used parts of the lwIP socket API thread safe. The complexity of this approach is yet to be determined. Can be combined with the task described in chapter 6.5.1.

6.6 Architectural viewpoints

The relation between viewpoints and views is analog to that between classes and objects. The viewpoints are often diagram types, while the views are specific diagrams that document a part of the architecture. This is much like the way an object is an instance of a class.

The described viewpoints are currently meant to address all stakeholders. The chosen viewpoints are behavioral and structural. The behavioral viewpoint is instantiated in the form of UML 2.0 communication diagrams and sequence diagrams. The structural viewpoint is instantiated in the form of directory listings. The UML diagrams do also serve as structural views, in that they list the files that compose each module.

Any inconsistencies between the directory listings and communication diagrams must according to the standard be documented within this chapter. At the time of writing there were no known inconsistencies between the views created in this project.

The single relevant view found within the lwIP documentation (see Figure 6) was not made according to a known viewpoint. It is therefore difficult to evaluate its consistency with the lwIP implementation. If the labels of the small rectangles in Figure 6 are intended to be function names, then all names given in EMAC are wrong according to the Linux port and lwIP 1.3.0 in general. The rest are the actual names of different lwIP functions. If the dotted line indicates data exchange between two threads, then Figure 6 documents how the Linux port TAP interface thread and the Tcpip thread interoperate.

6.6.1 UML diagrams

The main references for UML 2.0 syntax and semantics were [45] and [46]. The UML diagrams in this document do not show classes or objects, but C modules[3]. In some cases there are several source and/or header files that define a single module. Another important structural notation is the active object, which depicts an OS-level thread in these diagrams.

Generally the communication diagrams are used for architectural overview, and sequence diagrams are used to show details. In the communication diagrams the standard nested

[2] http://lwip.scribblewiki.com/LwIP_and_multithreading
[3] See term list

numbering scheme is used, although the readability of it is questionable. Whenever there are communication diagrams with self calls or complex nesting, there will also be an accompanying sequence diagram that shows the details better. Function arguments are only listed when they are relevant to the overall control flow.

6.6.2 Directory listings

A directory listing gives a reasonable overview of the directory structure, which is the only static grouping of modules in lwIP / PBMS. Textual descriptions for each directory will be used to augment the listings.

Interface documentation is currently not maintained outside of the source. When following the idiom of having a header file and source file with corresponding names make up a module, the header is in effect the interface documentation.

6.7 The architectural views of PBMS

The parts of the PBMS architecture that are documented here are data transmission and reception. These are two essential operations in the design and implementation of the priority scheme.

6.7.1 The behavioral views

6.7.1.1 Socket send

Three sequence diagrams that describe the complete chain of events from socket ***sendto*** to network transmission will be given first. The described scenario is then summarized in a communication diagram, which also relates the different interfaces to the layered model of the message stack.

This first diagram shows how the socket ***sendto*** operation is implemented in two distinct interfaces; the socket API and the Netconn API. In some aspects this is a source of complexity in PBMS, especially when the functionality of the Netconn API alone is not required. The Tcpip thread is structured to handle the Netconn API functions, as these have both external parts and internal parts. The internal parts are those operations that must be performed in the lwIP core, in mutual exclusion. The lwIP core is a loosely defined term that encompasses memory operations, manipulating network interfaces, and more. Several other functions also use the ***tcpip_apimsg*** function, which is further described in chapter 7.

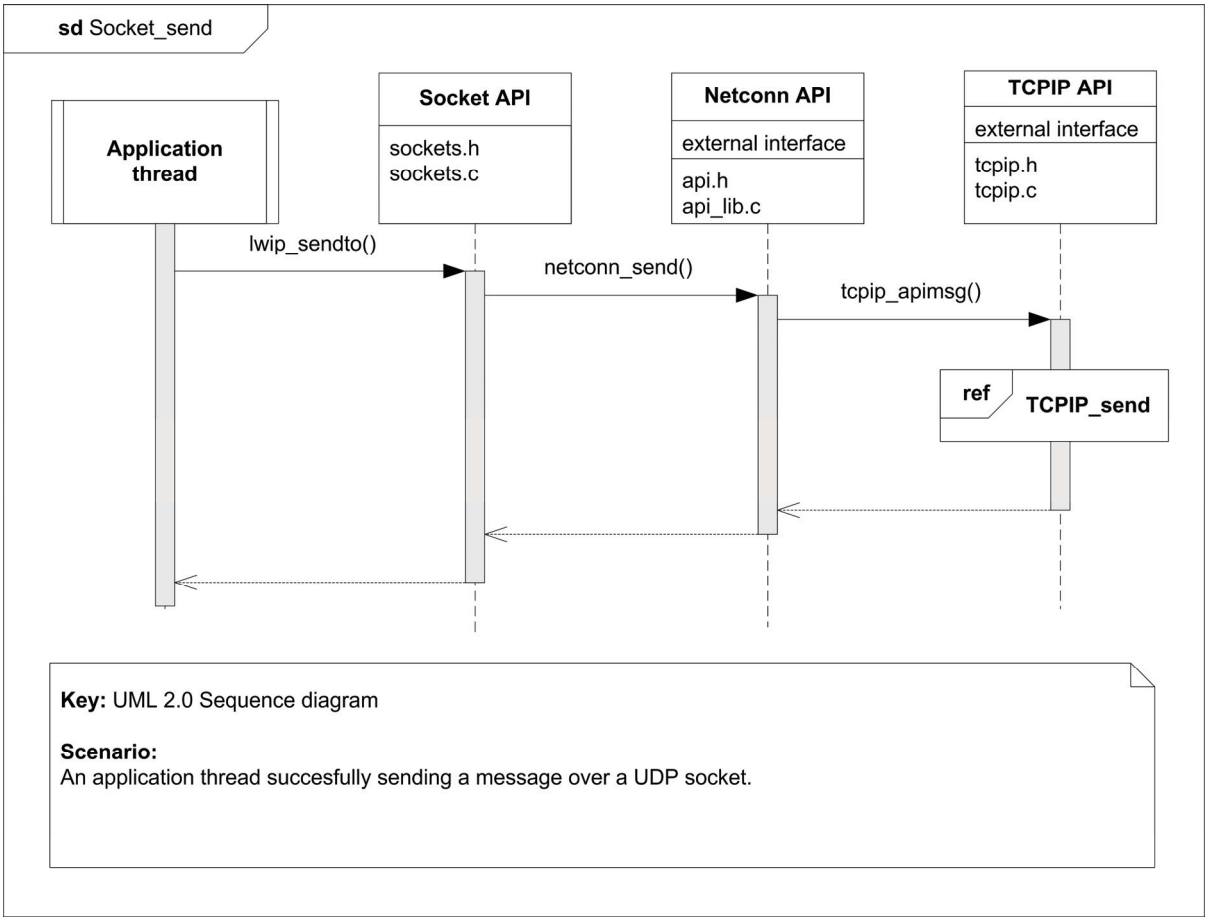

Figure 7: Socket send sequence diagram

The interactions between the Netconn API external interface, Tcpip thread and Netconn API internal interface are shown in this next diagram. The stack relies on message based synchronization of access to the protocol core. The Tcpip thread executes commands through function pointers in the messages it receives.

There is an ongoing discussion concerning the use of a stack-level mutex in the lwIP community. The task is not open to the public since some alternate unfinished socket code was published in it. The discussion concerning this task is available through lwip-devel archives [32], see message 2007-06/msg00095.html.

The priority queue mechanism of PBMS depends on the message based synchronization used in the current distribution of lwIP.

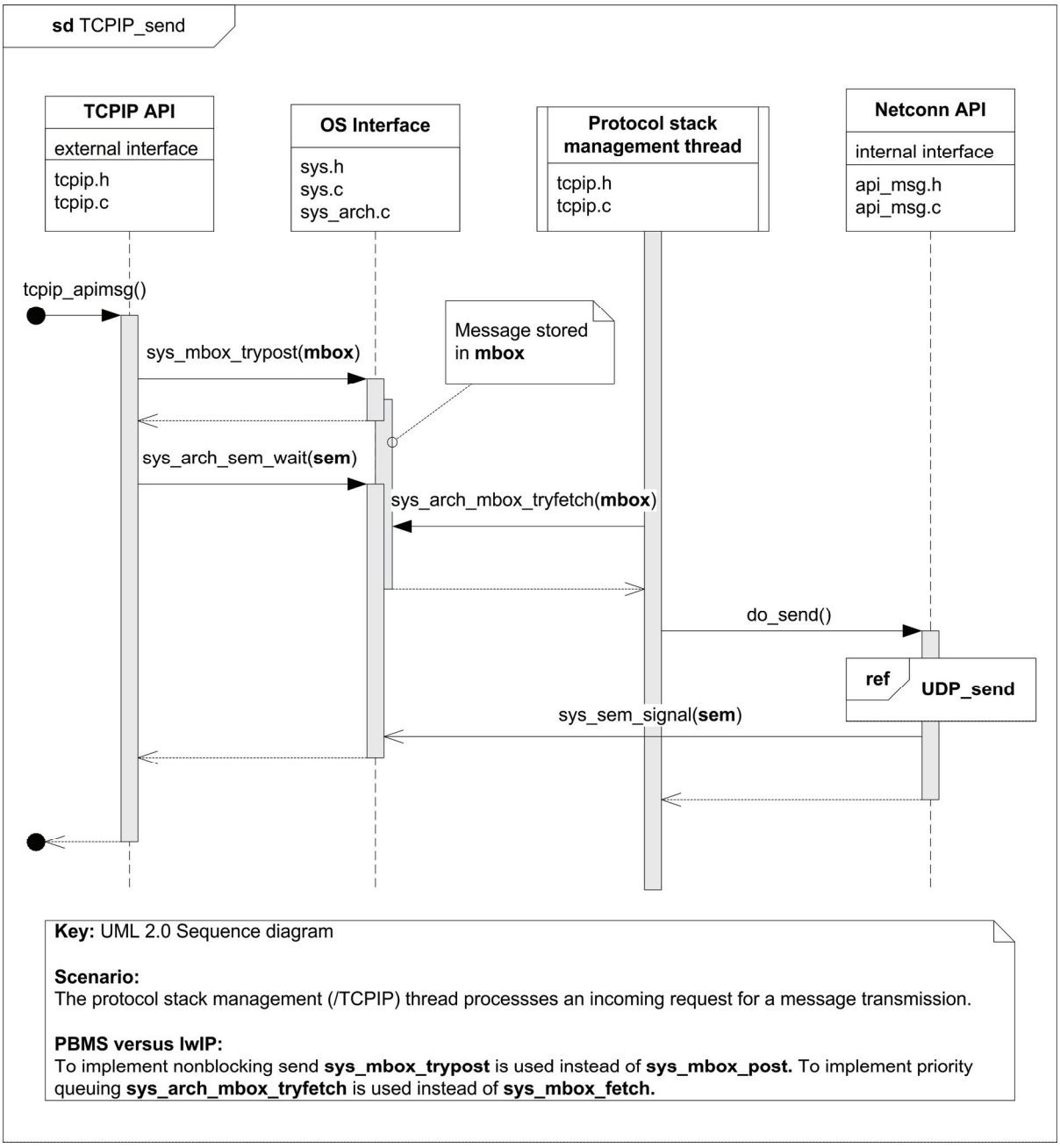

Figure 8: Tcpip send sequence diagram

In the priority scheme of PBMS ***tcpip_apimsg*** inserts the message in one of several message queues serviced by the Tcpip thread. The queue is chosen based on the TOS value stored in the calling threads protocol control block, as the IP packet has not been created yet. Upon retrieval the Tcpip thread executes the lower or internal part of the ***netconn_send*** function; ***do_send***. The nonblocking ***sendto*** function will return ***EWOULDBLOCK*** if the ***sys_mbox_trypost*** fails. Otherwise the function returns when a semaphore has been signaled from within the ***do_send*** function.

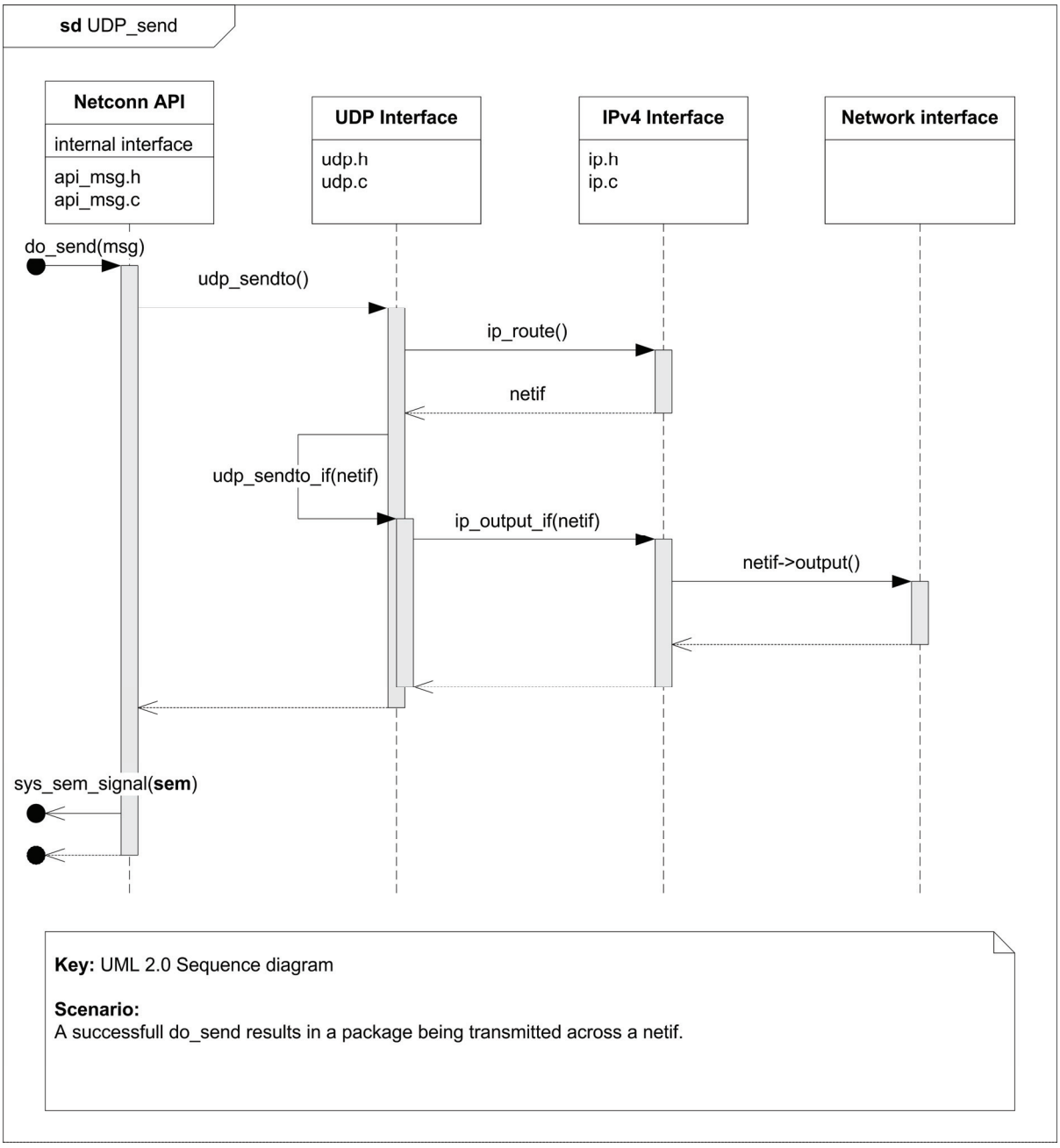

Figure 9: UDP send sequence diagram

This diagram shows the interaction with the UDP and IP interfaces. A specific network interface is first chosen, and with the found handle as argument ***ip_output_if*** leads to the message being transmitted.

PBMS data transmission (TX) communication diagram
Figure 11 shows a summary of the interactions described in the three preceding sequence diagrams.

The diagram details two threads of control:

- Application thread (prefix A)
- Tcpip thread (prefix B)

These threads interact via a message queue, referred to as ***mbox*** in Figure 11. The queue is one of several priority queues serviced by the Tcpip thread. The correct queue is chosen based on the TOS value stored in the calling threads protocol control block, as the IP packet has not been created yet.

The following sketch indicates the data flow from application to network:

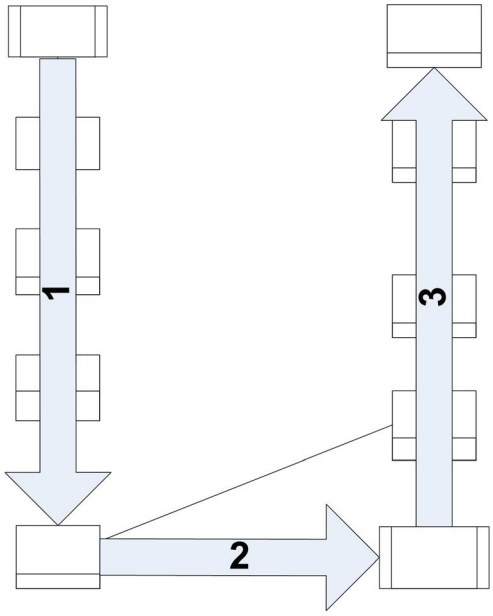

Figure 10: Data flow in Figure 11

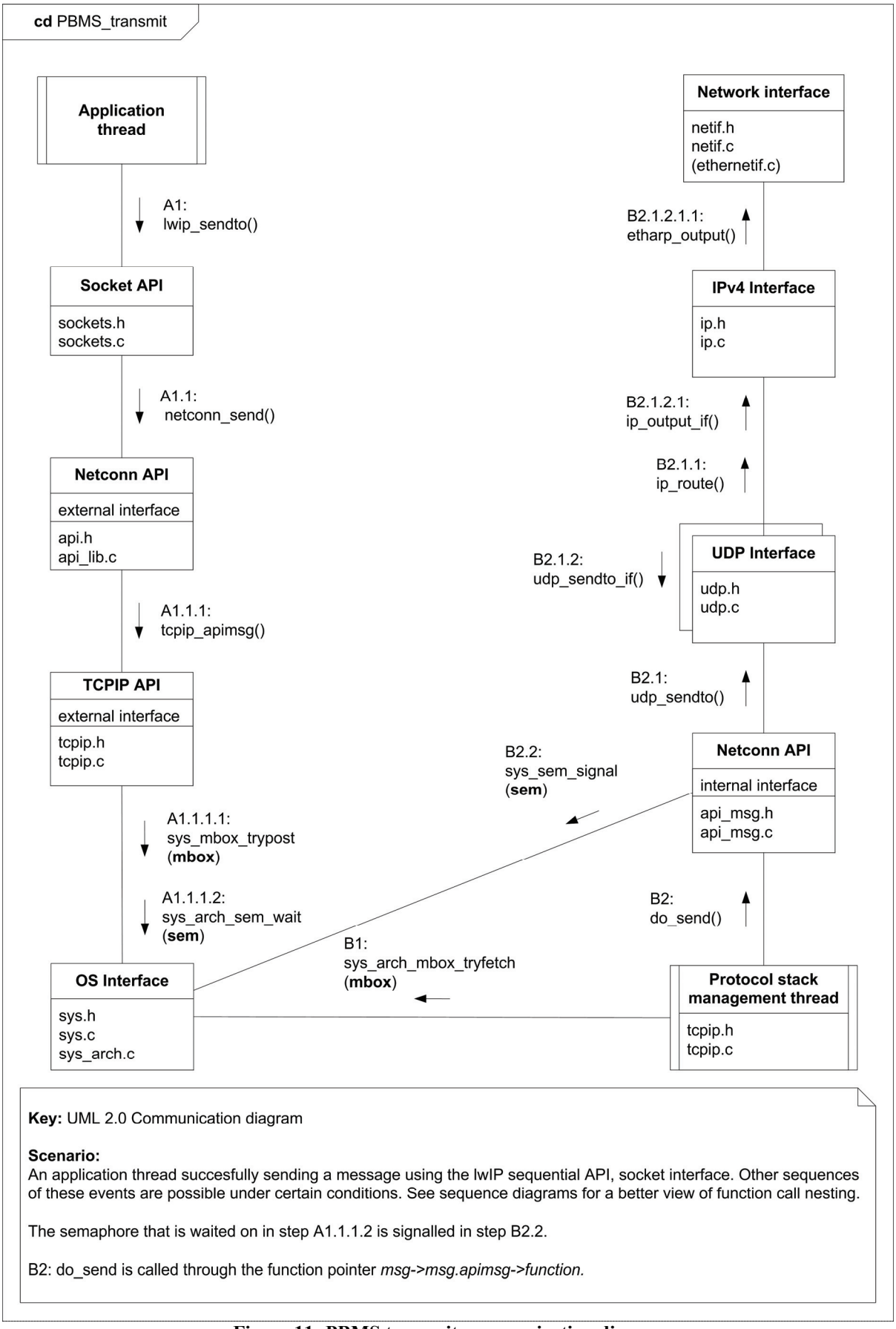

Figure 11: PBMS transmit communication diagram

6.7.1.2 Socket receive

The socket receive operation involves three threads; application, Tcpip and netif. The network interface activity can also be realized as interrupt service routines. Incoming data packets are queued according to the IPv4 header TOS field. The first diagram shown is the sequence from when the Tcpip thread fetches the newly arrived data packet from one of the priority queues, and ends when the packet is delivered to the UDP interface. There is an optional ARP update on incoming IP packets. This feature is discussed further in chapter 7.

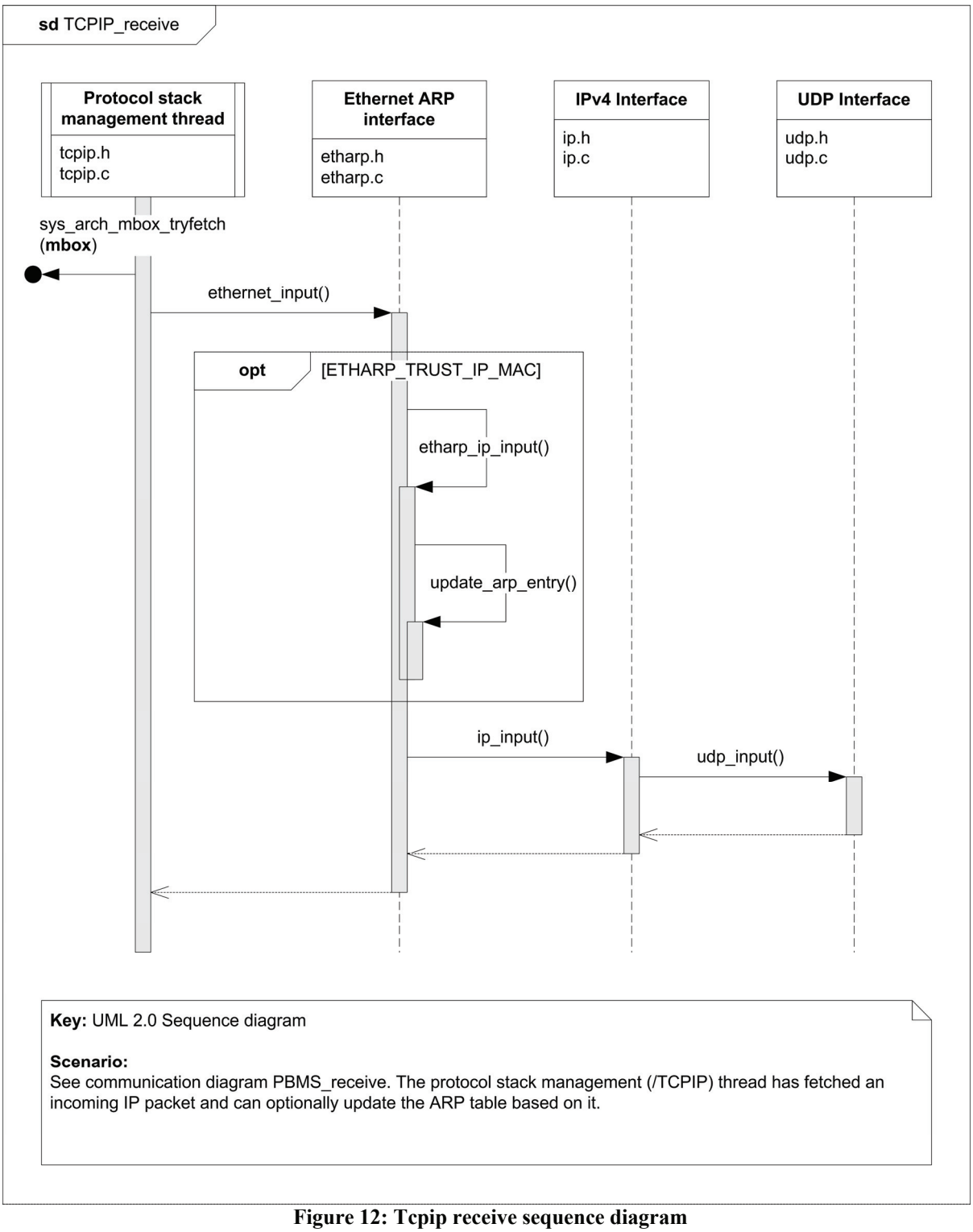

Figure 12: Tcpip receive sequence diagram

PBMS data reception (RX) communication diagram
The next diagram details how the three different threads interact via two different message queues in the message reception process.

The threads are:

- Application thread (prefix A)
- Tcpip thread (prefix B)
- Netif thread (prefix C)

The message queues the threads interface are:

- One of the priority queues serviced by the Tcpip thread (***mbox*** in Figure 14)
  Which queue the packet is inserted into by the Netif thread is determined by the TOS value found in the IPv4 header.
- The socket-specific receive message queue (***recvmbox*** in Figure 14)
  Each socket (technically, each UDP protocol control block) has an associated receive message queue.

There is no restriction on the interleaving of the application thread and Netif thread since the interactions are buffered. The following sketch indicates the data flow from network to socket:

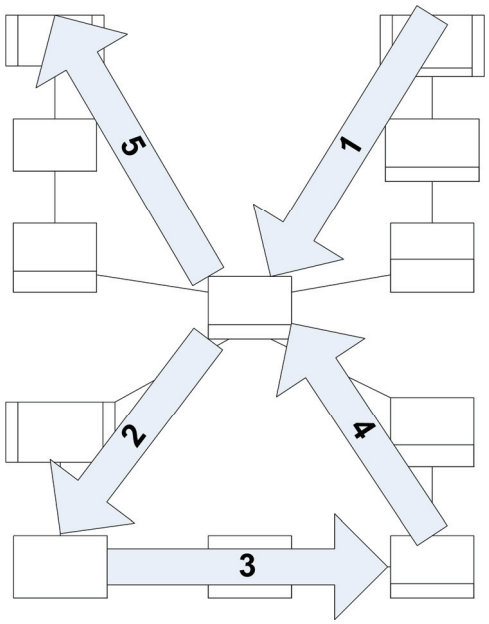

Figure 13: Data flow in Figure 14

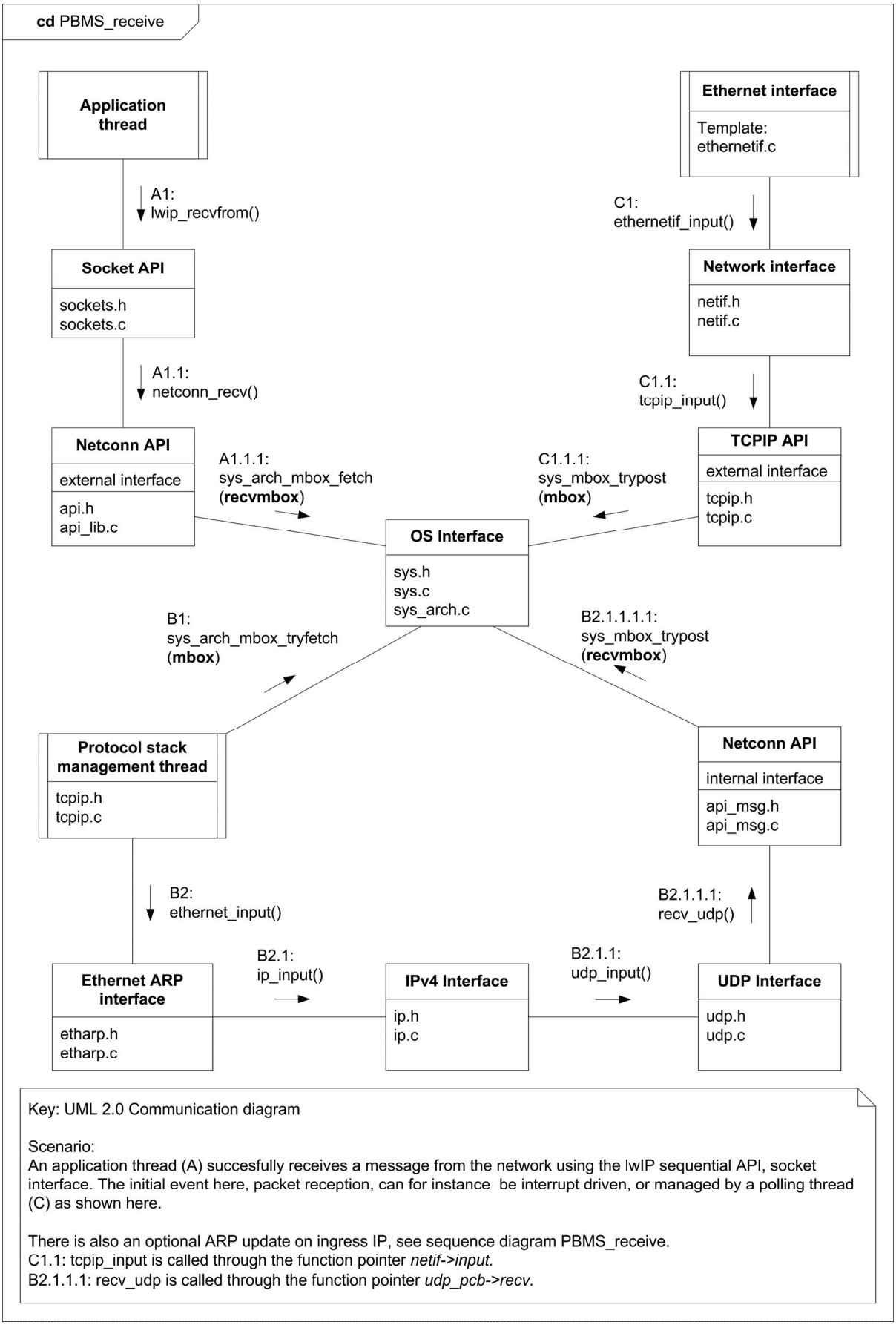

Figure 14: PBMS receive communication diagram

6.7.2 The structural views

There are three sets of files described in this chapter; the modified lwIP source, the Linux port and the files developed for PBMS. The main directory of the Linux port is named ***contrib***, brief for contributions. That is also the package name when downloading lwIP from the CVS repository. The Integrity port will probably be based on the Linux port, and the directory structure will have to be the same, unless the build process is altered. There are three files which make up the PBMS addition to lwIP, in addition to the modifications done to the lwIP source.

On the left below are the directories containing the lwIP and PBMS source code, on the right are the directories of the Linux port. Those lwIP directories not shown in bold (ipv6, SNMP and PPP) are not used in PBMS, and are therefore not described any further in this report.

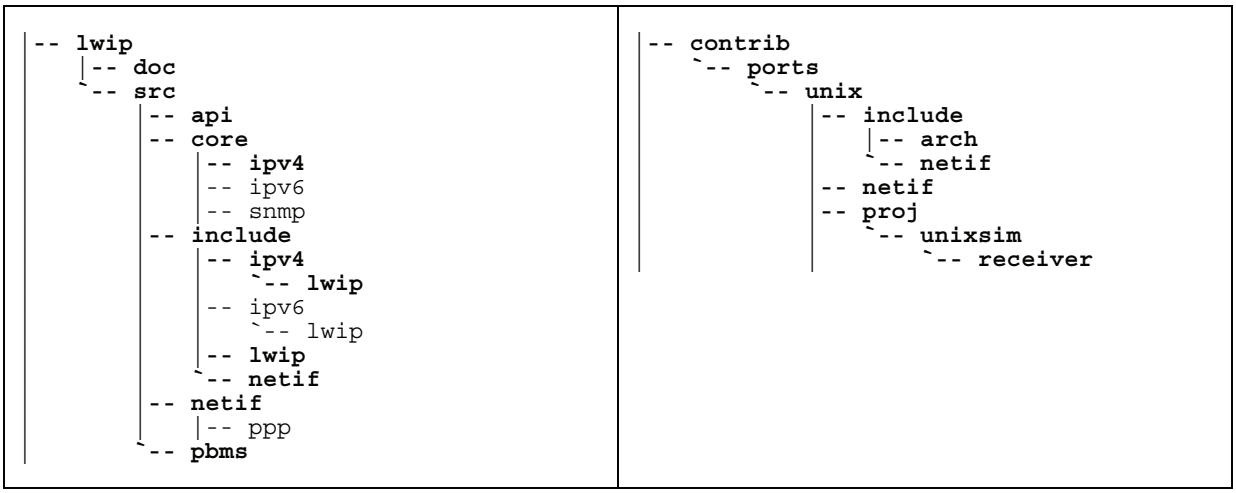

Table 4: lwIP and Linux port folder overview

There is some documentation distributed along with the lwIP source:

```
-- lwip
|-- CHANGELOG
|-- COPYING
|-- FILES
|-- README
|-- doc
|-- FILES
|-- contrib.txt
|-- rawapi.txt
|-- savannah.txt
|-- snmp_agent.txt
`-- sys_arch.txt
```

Most of these are specific to lwIP, but can be useful background information when working with PBMS. The file ***COPYING*** contains the BSD license lwIP is published under.

6.7.2.1 The PBMS files

The lwIP header files and source files are distributed in two separate directory trees. There is not a one-to-one mapping between the two trees, so both are detailed below. Almost all files follow the idiom of having header and source file pairs with equal names.

Those files where some of the offered functionality is used in PBMS are given in bold, although this markup may not be entirely accurate. A (*) means the original lwIP file is modified in PBMS, while a (+) indicates a new file or directory. The current build process must have all files available. Whether anything should be done about that is discussed in chapter 6.5.1.

```
|-- lwip
|   `-- src
|       |-- include
|       |   |-- ipv4
|       |   |   `-- lwip
|       |   |       |-- autoip.h
|       |   |       |-- icmp.h
|       |   |       |-- igmp.h
|       |   |       |-- inet.h
|       |   |       |-- inet_chksum.h
|       |   |       |-- ip.h
|       |   |       |-- ip_addr.h
|       |   |       `-- ip_frag.h
|       |   `-- lwip
|       |       |-- api.h
|       |       |-- api_msg.h
|       |       |-- arch.h
|       |       |-- debug.h *
|       |       |-- def.h
|       |       |-- dhcp.h
|       |       |-- dns.h
|       |       |-- err.h *
|       |       |-- init.h
|       |       |-- mem.h
|       |       |-- memp.h
|       |       |-- memp_std.h
|       |       |-- netbuf.h
|       |       |-- netdb.h
|       |       |-- netif.h
|       |       |-- netifapi.h
|       |       |-- opt.h
|       |       |-- pbuf.h
|       |       |-- raw.h
|       |       |-- sio.h
|       |       |-- snmp.h
|       |       |-- snmp_asn1.h
|       |       |-- snmp_msg.h
|       |       |-- snmp_structs.h
|       |       |-- sockets.h *
|       |       |-- stats.h
|       |       |-- sys.h
|       |       |-- tcp.h
|       |       |-- tcpip.h
|       |       `-- udp.h
|       |   `-- netif
|       |       |-- etharp.h
|       |       |-- loopif.h
|       |       |-- ppp_oe.h
|       |       `-- slipif.h
```

```
|-- lwip
|   `-- src
|       |-- FILES
|       |-- api
|       |   |-- api_lib.c *
|       |   |-- api_msg.c *
|       |   |-- err.c *
|       |   |-- netbuf.c
|       |   |-- netdb.c
|       |   |-- netifapi.c
|       |   |-- sockets.c *
|       |   `-- tcpip.c *
|       |-- core
|       |   |-- dhcp.c
|       |   |-- dns.c
|       |   |-- init.c
|       |   |-- ipv4
|       |   |   |-- autoip.c
|       |   |   |-- icmp.c
|       |   |   |-- igmp.c
|       |   |   |-- inet.c
|       |   |   |-- inet_chksum.c
|       |   |   |-- ip.c *
|       |   |   |-- ip_addr.c
|       |   |   `-- ip_frag.c
|       |   |-- mem.c
|       |   |-- memp.c
|       |   |-- netif.c
|       |   |-- pbuf.c
|       |   |-- raw.c
|       |   |-- stats.c
|       |   |-- sys.c
|       |   |-- tcp.c
|       |   |-- tcp_in.c
|       |   |-- tcp_out.c
|       |   |-- udp.c
|       |-- netif
|       |   |-- FILES
|       |   |-- etharp.c
|       |   |-- ethernetif.c
|       |   |-- loopif.c
|       |   `-- slipif.c
|       `-- pbms +
|           |-- bootp.c +
|           |-- bootp.h +
|           `-- priorities.h +
```

Table 5: The PBMS source files

New files in PBMS
The directory ***pbms*** contains the files that were developed in this project, and is not part of lwIP. The file *priorities.h* is where the priority scheme of PBMS is configured. How that is done is described in the user documentation, see chapter 3. The PBMS BOOTP implementation is described in chapter 7.5.

The layered architecture of lwIP
How the most important interfaces in the lwIP source map onto the TCP/IP layered architecture is outlined in Table 6. The Netconn API can be interpreted as a separate user interface in the application layer, but in PBMS it is only used as an internal module in the sockets API. This is also true for the Tcpip API.

Application layer BSD Sockets API → sockets.h, sockets.c	**Stack management** Netconn API → api.h, api_lib.c (external interface) → api_msg.h, api_msg.c (internal interface) Tcpip API and thread → tcpip.h, tcpip.c **OS abstraction** Threads, semaphores, message boxes → sys.h, sys.c Memory → mem.h, mem.c → memp.h, memp.c, memp_std.h
Transport layer UDP → udp.h, udp.c	
Network layer IPv4 → ip.h, ip.c, → inet.h, inet.c → ip_addr.h, ip_addr.c	
Data link layer ARP → etharp.h, etharp.c Generic network interface → netif.h, netif.c	

Table 6: The layered architecture of lwIP

6.7.2.2 The Linux port files

In the following table (*) means the original Linux port file is modified in the PBMS Linux platform. A (+) indicates a new file or directory.

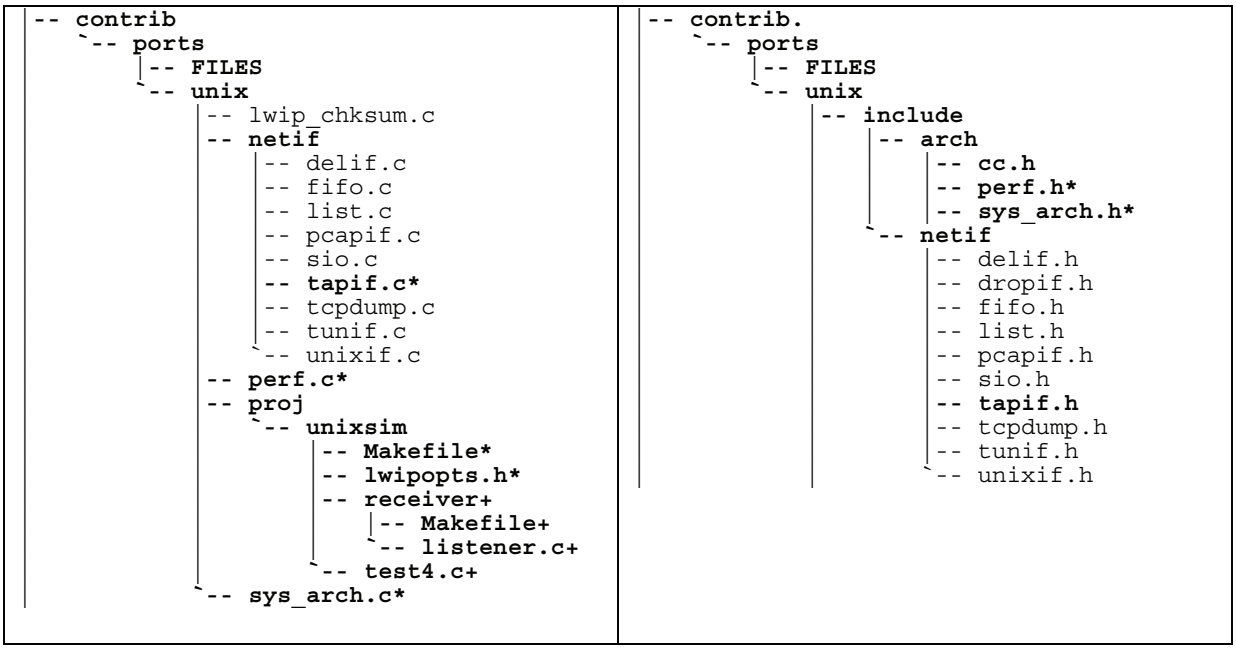

Table 7: The lwIP Linux port files

Those files that were used in the Linux test platform of PBMS are shown in bold. These can be seen as a platform specific extension of the OS abstraction layer in Table 6.

7 The PBMS Design and Implementation

7.1 Introduction

This chapter complements the architectural description of chapter 6 with detailed descriptions of some key functions and data structures in PBMS. It is also discussed how these fulfill the requirements given in chapter 4, either directly through lwIP components, or through the additional functionality of PBMS.

According to IEEE standard 1016 [47] a software design description (SDD) should give a uniform description of a set of entities that make up a program. The last chapter gave an overview of the directory structure, protocol interfaces and cross-interface interactions. The entities described in this chapter will therefore be on a lower level, with descriptions of some of the key functions and data structures in PBMS.

The entities shall according to the standard be described according to ten different attributes, although some may have the value *none* when not applicable. The attributes are shown with a bold typeface in the following paragraphs, with a subscript indicating the relevant paragraph in the standard.

Each function and data structure can be uniquely **identified**$_{5.3.1}$ using their name. Considering that some functions may be static, the files where they are declared and defined are also given. Most of the code base is implemented as C modules[4]. The **types**$_{5.3.2}$ of entities has already been noted; functions and structs written in ANSI C. The **purpose**$_{5.3.3}$ and **function**$_{5.3.4}$ of each entity will be stated when this was well documented in lwIP, or where such information has been extracted in this project. Detailing **subordinates**$_{5.3.5}$ will only be applicable to structs, as there are no internal or anonymous functions in the code base.

A function-entity will, disregarding error handling, **depend**$_{5.3.6}$ on those other functions that it calls. According to the standard this shall be documented in the form of the *uses* relation or *requires the presence of*-relation. These are generally not documented in lwIP, as discussed in chapter 5.5.3. One could simply list all function calls that are within the described function-entity. That would not be accurate considering that a variable amount of error handling is done throughout the code base.

A function-entity will have a well-defined **interface**$_{5.3.7}$, easily accessible through the available source code. Interfaces are only described in complete detail when such information is relevant. For struct-entities this is mainly to describe data formats. As mentioned in chapter 5.5.4 there is an up to date code reference for lwIP available at the lwIP ScribbleWiki [5].

The described **resources**$_{5.3.8}$ is message queues, semaphores and the other facilities offered by the OS abstraction layer. The logical operation of a function is referred to as **processing**$_{5.3.9}$, and the internal **data**$_{5.3.10}$ it uses is also to be detailed.

This chapter is not a complete SDD according to the standard, since not all entities are documented, and information on some key attributes is lacking. The stated purpose is to give a description of some important functions and data structures, with emphasis on the essential differences between PBMS and lwIP.

[4] See term list

7.2 Requirements met directly by lwIP

All requirements referred to are given in chapter 4 of this report. All requirements that are supposedly met *off-the-shelf* are to be validated in a target system test. See chapter 8 for an outline of how this will be done. LwIP is also discussed in light of Nera's design considerations as presented in chapter 5.3.

If nothing else is stated the lwIP version used is 1.3.0-stable, including the bug fixes up until 2008-05-09.

7.2.1 Software interface requirements

LwIP implements IPv4, UDP and ARP, satisfying the essential requirements PBMS1210, PBMS1230 and PBMS1240. The ARP implementation is according to the code documentation [5] provided by Leon Woestenberg from the company Axon Digital Design B.V. It is stated that it complies with RFC 826, and also supports gratuitous ARP from RFC3220.

LwIP offers a partial implementation of the BSD socket interface, satisfying the conditional requirement PBMS1300.

The TOS value for a socket could be set using ***setsockopt*** and read using ***getsockopt***. The value was also properly assigned to outgoing packets. It was possible to disable checksums with a compile flag. Both the proper TOS labeling and checksum disabling was checked by examining the transmitted packets with Wireshark [48].

7.2.2 Performance requirements

According to [4], quoted below, lwIP fulfills requirement PBMS1510. This claim is however not related to a compilation for a specific architecture, as specified in the requirement. This claim is also presented at the ScribbleWiki, which can be edited by anyone. It is therefore not considered to be reliable information, and will not be given any weight in the verification of PBMS.

> *"The focus of the lwIP TCP/IP implementation is to reduce resource usage while still having a full scale TCP. This making lwIP suitable for use in embedded systems with tens of kilobytes of free RAM and room for around 40 kilobytes of code ROM."*

The code size of PBMS will be verified the target system. PBMS does not use the full scale TCP offered by lwIP, so a certain reduction in code size could actually be expected. Minimizing the binary size of PBMS was not a primary goal during this phase of the implementation, so there are probably several potential optimization points.

7.2.3 Process requirements

LwIP is available under a BSD license [4], which satisfies requirement PBMS1800 in chapter 4.3.

By compiling the example applications in the Linux port with the gcc *–ANSI* flag, I found that the lwIP source met requirement PBMS1700. The test platform had a few non-compliant features. The only thing that must be removed is the use of ***struct timezone*** in sys_arch.c.

7.3 The Tcpip thread and associated APIs

PBMS has one thread that takes care of all input (except for a few operations handled by the Netif thread), output and internal processing. This is called the Tcpip thread.
Function name: ***static void tcpip_thread(void *arg)***
File(s): **tcpip.c**

The Tcpip thread's mode of operation is to fetch a single message from one of the priority queues and execute an operation through a function pointer in the message. This central point of control for both incoming and outgoing traffic was a key enabler for the relatively simple implementation of the priority scheme.

First the PBMS packet input scheduling is discussed. Then some of the general socket and Netconn API functionality is explained. A basic understanding of those features is required before the PBMS output priority scheme can be described. After that the socket receive prioritization is presented. This uses the same API mechanisms as the PBMS output scheme.

The type-of-service bits in the IPv4 header [14], is in RFC 2474 [20] reinterpreted as the DiffServ Code Point (DSCP). The lwIP stack already had a proper implementation of setting and getting the TOS field in the IP header through socket system calls. I therefore decided to go with using the (obsolete) TOS field. If it desirable to use DSCP values instead, that will be a fairly limited modification of the code base.

7.3.1 Strict priority input scheduling

The fundamental difference between lwIP and PBMS is that incoming and outgoing traffic in PBMS is scheduled according to priority. Strict priority scheduling operates according to a user defined mapping between TOS precedence values in the IP layer and a user-defined number of separate message queues.

7.3.1.1 Incoming packets – original lwIP code

Packet insertion is done by the network interface thread.

```
if (sys_mbox_trypost(mbox, msg) != ERR_OK) {
  memp_free(MEMP_TCPIP_MSG_INPKT, msg);
  return ERR_MEM;
}
return ERR_OK;
```

Table 8: Original lwIP source: From tcpip_input in tcpip.c

Packet extraction done by Tcpip thread:

```
while (1) {                          /* MAIN Loop */
  sys_mbox_fetch(mbox, (void *)&msg);
  switch (msg->type) {
```

Table 9: Original lwIP source: From tcpip_thread in tcpip.c

7.3.1.2 Incoming packets – PBMS code

```
/* INPUT TOS peek */
ip_header = p->payload;
tos = IPTOS_PREC( IPH_TOS(ip_header) );
input_queue = -1;
switch(tos){
  case IPTOS_PREC_NETCONTROL:
    input_queue = PBMS_INPUT_PREC_NETCONTROL;
    break;
  case IPTOS_PREC_INTERNETCONTROL:
    input_queue = PBMS_INPUT_PREC_INTERNETCONTROL;
    break;
  case IPTOS_PREC_CRITIC_ECP:
    input_queue = PBMS_INPUT_PREC_CRITIC_ECP;
    break;
  case IPTOS_PREC_FLASHOVERRIDE:
    input_queue = PBMS_INPUT_PREC_FLASHOVERRIDE;
    break;
  case IPTOS_PREC_FLASH:
    input_queue = PBMS_INPUT_PREC_FLASH;
    break;
  case IPTOS_PREC_IMMEDIATE:
    input_queue = PBMS_INPUT_PREC_IMMEDIATE;
    break;
  case IPTOS_PREC_PRIORITY:
    input_queue = PBMS_INPUT_PREC_PRIORITY;
    break;
  case IPTOS_PREC_ROUTINE:
    input_queue = PBMS_INPUT_PREC_ROUTINE;
    break;
  default:
    LWIP_ASSERT("PBMS: improper priority assignment to incoming packet", 0 );
    break;
}

LWIP_ASSERT("PBMS: Proper priority assignment", (input_queue>=0) );

/* Packet drop scheduling: Keep the old */
if (sys_mbox_trypost(mboxes[input_queue], msg) != ERR_OK) {
  memp_free(MEMP_TCPIP_MSG_INPKT, msg);
  return ERR_MEM;
}
return ERR_OK;
```

Table 10: PBMS : From tcpip_input in tcpip.c

Note: Some code is skipped for clarity. In the delivered source ***i*** is named ***priority_level***.

Packet insertion done by Netif thread:

```
while (1) {                          /* MAIN Loop */
  msg_fetched = 0;
  /* PBMS: Strict priority queueing */
  for( i = PBMS_PRIORITY_LEVELS-1; i>=0 ; i-- ){
    if( sys_arch_mbox_tryfetch( mboxes[i], (void *)&msg ) != SYS_MBOX_EMPTY ){
      msg_fetched= 1;
      break;
    }
  }

  if(msg_fetched == 0){
    continue;
  }

  switch (msg->type) {
```

Table 11: PBMS: From tcpip_thread in tcpip.c

One notable difference between PBMS (Table 11) and lwIP (Table 9) is that the Tcpip thread will now continuously poll the queues.

In PBMS the packets are inserted into the appropriate queue in the function ***tcpip_input()***, defined in tcpip.h/tcpip.c. (See Table 10) By interpreting the ***p->payload*** argument as an IP header the TOS-peek can be carried out. This is written for the Linux port, TAP interface, and depends on only receiving IP packets - see discussion on ARP update for ingress IP in chapter 7.4.2.

The included code also shows the currently implemented packet drop strategy; drop new packets in the event that the queue is full. According to the design consideration in chapter 5.3.3, PBMS should at least support a compile option of dropping the oldest queued messages instead.

The queue size is currently equal for all message boxes, but the option ***LWIP_SO_RCVBUF*** can be used to limit the size of only the receive queues. This option is realized through the integer variable ***recv_bufsize*** in ***struct netconn***, defined in api.h. The value defines the maximum amount of bytes that can be queued. It is currently defaulted to ***INT_MAX*** when enabled, but it can easily be made configurable from lwipopts.h.

7.3.1.3 Incoming packets – Configurable prototype

To do a basic test of a scheme that could be configurable at run time I implemented the priority assignment through a table lookup. In the same branch of the source I updated the ARP handling to be in line with the normative lwIP 1.3.0 design, and let ***tcpip_input*** process the entire Ethernet frame. This branch did not work with the Linux port, so I reverted to the approach shown above for all the tests that have been performed.

The exact problem with this code was not resolved due to a lack of time. It was tested with the Linux TAP interface. The ARP replies sent to PBMS were equal for this code and the one that works, but the stack did not seem to process the incoming packet properly.

The extraction in the Tcpip thread is similar to the one showed in Table 11, only the insertion is different. As discussed in chapter 6.4.2, using an array as shown in Table 12 is the first step towards a run-time configurable priority scheme.

```
/* Tcpip thread receives Ethernet frames according to lwIP architecture */
switch (mode){
  case TCPIP_INPUT_ETHERNET:
    ethhdr = p->payload;
    switch (ntohs(ethhdr->type)){
      case ETHTYPE_IP:
        ethiphdr = p->payload;
        tos = ( ( ntohs(ethiphdr->ip._v_hl_tos) & 0xff ) & IPTOS_PREC_MASK );
        tos = (tos >> 5);
        input_queue = tos_to_input_q[tos];
        break;
      case ETHTYPE_ARP:
        input_queue = PBMS_INPUT_ARP_PRI;
        break;
      default:
        LWIP_ERROR("Error in packet header" ,1 , return ERR_MEM;);
        break;
    }
    break;

    default:
      LWIP_ERROR("Error in input mode" ,1 , return ERR_MEM; );
      break;
  }
```

Table 12: Discontinued branch of PBMS: From tcpip_input in tcpip.c

7.3.2 The Netconn API and the Tcpip thread

PBMS has three conceptual API layers; the BSD sockets API, the Netconn API and the Tcpip thread API. This chapter describes how the Netconn API and Tcpip thread function together, to lay the ground for the description of the priority scheme in the coming subchapters. The external Netconn API posts messages to the Tcpip thread via ***tcpip_apimsg***. The Tcpip thread then executes an internal Netconn API function.

The Tcpip thread input message is defined by the ***struct tcpip_msg***, defined in tcpip.h. The thread recognizes five classes of input messages, with the associated flag in parentheses:

- API messages (***TCPIP_MSG_API***)
- Input packet (***TCPIP_MSG_INPKT***) (Used by ***tcpip_input***)
- Callback (***TCPIP_MSG_CALLBACK***)
- Timeout (***TCPIP_MSG_TIMEOUT***)
- Netif API message (***TCPIP_MSG_NETIFAPI***) (Currently not used in PBMS)

7.3.2.1 API messages

The API messages are used to execute the parts of Netconn API functions that require mutually exclusive access to the lwIP core. The table below shows the functions related to UDP and ARP. Lower function parts (***do_ ****) are found in api_msg.c, and the Netconn API is found in api_lib.c.

One of these interactions, executing ***netconn_send*** and ***do_send***, is shown in detail in Figure 7 and Figure 8, see chapter 6. All pairs given below are executed in the same manner, except those ***do_*** functions that do not use the ***ACK*** macro. Those do not use the semaphore for end synchronization.

The two macros referred to in the table are defined in tcpip.h:
#define TCPIP_APIMSG(m) ***tcpip_apimsg(m)***
#define TCPIP_APIMSG_ACK(m) ***sys_sem_signal(m->conn->op_completed)***

Netconn API function:	**Uses macro:**	**Lower function part: (Executed by Tcpip thread)**	**do_ contains ACK macro?**
netconn_new_with_proto_and_callback	Yes	***do_newconn***	Yes
netconn_delete	**No**	***do_delconn***	**No**
netconn_getaddr	Yes	***do_getaddr***	Yes
netconn_bind	Yes	***do_bind***	Yes
netconn_connect	**No**	***do_connect***	**No**
netconn_disconnect	Yes	***do_disconnect***	Yes
netconn_recv	Yes	***do_recv***	Yes
netconn_send	Yes	***do_send***	Yes

Table 13: External and internal functions in the Netconn API

7.3.2.2 Timeout

With the current system configuration the timeout signal is only used to handle the ARP timer. Functions called are ***sys_timeout()*** and ***sys_untimeout(),*** see sys.h / sys.c. It is not recommended to use these when a platform-specific timer is available.

7.3.3 Strict priority output and receive scheduling

LwIP already implemented a proper initialization of the IPv4 TOS field based on the socket TOS. What has been implemented in PBMS is to prioritize packet streams internally in the stack based on the socket TOS.

7.3.3.1 Scheduling of API messages from *netconn_send*

The interaction pattern between the application thread and Tcpip thread for sending a message has been described in the previous chapter. In order to prioritize different streams of outgoing traffic in a manner that mirrors the incoming traffic, the API messages that trigger ***do_send*** (i.e. contain a function pointer to ***do_send***) are in PBMS given specific priorities according to the TOS of the sending protocol control block (PCB):

tos = IPTOS_PREC(apimsg->msg.conn->pcb.udp->tos);

Besides the altered TOS peek the code is similar to the input scheduling.

7.3.3.2 Scheduling of API messages from *netconn_recv*

A request for receiving a message from the network is handled by the Tcpip thread. To maintain an unbroken chain of priorities this request too has to be prioritized according to the TOS of the calling socket.

7.3.3.3 Potential refactoring of priority queue

No profiling has of yet been done, so there is currently no reason to implement what is described here. If PBMS is profiled on the target OS, and it is determined that the priority queue module is ineffective, one possible solution is to implement another data structure. The current queue implementation is a linked list.

Rönngren and Ayani did a comparative study of parallel and sequential priority queue algorithms in [49]. The requirement for PBMS would be a sequential algorithm with the best worst-case performance. The maximum queue size is a compile time option, and will probably be kept below 1000 elements, which is a differentiating factor in the study. The two algorithms with best overall worst-case performances were:

- Splay tree, a heuristically balanced binary search tree.
- Skew heap, a heap-ordered binary tree.

The Splay tree is stable, while the Skew heap is not. Stability means that a FIFO ordering of events on the same priority level is maintained. This would be a desired feature in PBMS, so the Splay tree is the preferred choice of these two. The authors also considered which algorithms that would be suitable for hard real time systems. Table 14 details the expected asymptotic performances of two implementations from the study.

Queue	**Enqueue amortized (expected, worst-case)**	**Enqueue max**	**Dequeue amortized (expected, worst-case)**	**Dequeue max**
Splay tree	O(log(n)), O(log(n))	O(n)	O(1), O(1)	O(1)
Implicit Binary Heap	O(1), O(log(n))	O(log(n))	O(log(n)), O(log(n))	O(log(n))

Table 14: Expected performance of two sequential priority queues.

Based on the experimental results Rönngren and Ayani recommend the Implicit Binary Heap for hard real time systems, since it is the only algorithm that has a worst-case access time running in less than O(n).

7.4 Network interface thread

7.4.1 The TAP interface thread in the Linux port

The reception of data from the network interface is handled by a separate thread in the Linux port of lwIP; see Figure 14 (thread C). This thread has a relatively simple mode of operation.
Function: ***static void tapif_thread(void *arg)***
File(s): tapif.c (Linux port)

```
while(1) {
  FD_ZERO(&fdset);
  FD_SET(tapif->fd, &fdset);

  /* Wait for a packet to arrive. */
  ret = select(tapif->fd + 1, &fdset, NULL, NULL, NULL);

  if(ret == 1) {
    /* Handle incoming packet. */
    tapif_input(netif);
  } else if(ret == -1) {
    perror("tapif_thread: select");
  }
}
```

Table 15: Network interface thread in Linux port

The ARP handling of the Linux port is defined in ***tapif_input***.
Function: ***static void tapif_input(struct netif *netif)***
File(s): tapif.c (Linux port)

```
static void
tapif_input(struct netif *netif)
{
  struct tapif *tapif;
  struct eth_hdr *ethhdr;
  struct pbuf *p;

  tapif = netif->state;
  p = low_level_input(tapif);
  ethhdr = p->payload;

  switch(htons(ethhdr->type)) {
  case ETHTYPE_IP:
#if 0
/* CSi disabled ARP table update on ingress IP packets.
   This seems to work but needs thorough testing. */
    etharp_ip_input(netif, p);
#endif
    pbuf_header(p, -14);
    netif->input(p, netif);
    break;
  case ETHTYPE_ARP:
    etharp_arp_input(netif, tapif->ethaddr, p);
    break;
  default:
    pbuf_free(p);
    break;
  }
}
```

Table 16: tapif_input from Linux port

7.4.2 ARP update on ingress IP in lwIP

The code in Table 16 shows that the Ethernet header is stripped, and only the IP packet is passed on to ***tcpip_input***. This code is compatible with the segment in Table 10, but not the code shown in Table 12.

A prototype for how the network interface input operation should be done is distributed with the lwIP source.
Function: ***static void ethernetif_input(struct netif *netif)***
File: ethernetif.c

```
static void
ethernetif_input(struct netif *netif)
{
  struct ethernetif *ethernetif;
  struct eth_hdr *ethhdr;
  struct pbuf *p;

  ethernetif = netif->state;

  /* move received packet into a new pbuf */
  p = low_level_input(netif);
  /* no packet could be read, silently ignore this */
  if (p == NULL) return;
  /* points to packet payload, which starts with an Ethernet header */
  ethhdr = p->payload;

  switch (htons(ethhdr->type)) {
  /* IP or ARP packet? */
  case ETHTYPE_IP:
  case ETHTYPE_ARP:
#if PPPOE_SUPPORT
  /* PPPoE packet? */
  case ETHTYPE_PPPOEDISC:
  case ETHTYPE_PPPOE:
#endif /* PPPOE_SUPPORT */
    /* full packet send to tcpip_thread to process */
    if (netif->input(p, netif)!=ERR_OK)
     { LWIP_DEBUGF(NETIF_DEBUG, ("ethernetif_input: IP input error\n"));
       pbuf_free(p);
       p = NULL;
     }
    break;

  default:
    pbuf_free(p);
    p = NULL;
    break;
  }
}
```

Table 17: ethernetif_input from the lwIP source

The whole Ethernet frame will then be processed in the Tcpip thread. Now have a look at Figure 12 in chapter 6, where the optional ARP update is outlined. The ***ETHARP_TRUST_IP_MAC*** option which enables this feature is given in the file lwipopts.h.

What seems to have happened is that the ARP handling was altered in version 1.3 of lwIP, but the Linux port was not updated accordingly. The motivation behind handling ARP packets in the Tcpip thread was to protect the ARP table from concurrent access, see discussion at [8] (2008-05/msg00006.html).

7.4.3 Modification: Process ARP in network interface thread

There is a potential performance gain in handling ARP packets in the network interface (Netif) thread instead of passing them up to the TCPIP thread. Parallel processing is a well-known tactic for increased performance [1]. The ARP table would then have to be protected from concurrent access, for instance by a semaphore.

In the priority scheme of PBMS one must do an explicit prioritization of everything that is to be handled by the Tcpip thread. Nera believed that the ARP traffic would have to be given top priority, and that will increase the worst-case delay time for all of the packet scheduling.

It would be possible to give relative priority to either the Netif thread or the Tcpip thread by giving them different OS level priorities. The default in lwIP is to have these run at equal priorities.

The final evaluation of this issue will be done when implementing the Integrity port of PBMS.

7.5 DHCP and BOOTP

7.5.1 The lwIP DHCP client

The interaction between the PBMS stack and the BOOTP server module was implemented in the same manner as the lwIP DHCP client. Therefore the DHCP module is described here in some detail. The network interface abstraction module (netif.h) has a struct holding the DHCP client state.

In ***ip_input()*** the following fragment is used:

```
#if LWIP_DHCP
if (netif == NULL) {
  if (IPH_PROTO(iphdr) == IP_PROTO_UDP) {
    if (ntohs(((struct udp_hdr *)((u8_t *)iphdr + iphdr_hlen))->dest)
        == DHCP_CLIENT_PORT)
      {
      netif = inp;
      check_ip_src = 0;
    }
  }
}
#endif /* LWIP_DHCP */
```

Table 18: DHCP code in ip_input (lwIP)

The function ***ip_input()*** examines the UDP header destination port to determine if this is in fact a proper DHCP message. After this piece of code the packet is passed on to ***udp_input()***.

```
(132) #if LWIP_DHCP

  pcb = NULL;
  if (dest == DHCP_CLIENT_PORT) {
    if (src == DHCP_SERVER_PORT) {
      if ((inp->dhcp != NULL) && (inp->dhcp->pcb != NULL)) {
        if ((ip_addr_isany(&inp->dhcp->pcb->remote_ip) ||
           ip_addr_cmp(&(inp->dhcp->pcb->remote_ip), &(iphdr->src)))) {
          pcb = inp->dhcp->pcb;
        }
      }
    }
  } else

(150) #endif /* LWIP_DHCP */

(…)

(277) pcb->recv(pcb->recv_arg, pcb, p, &(iphdr->src), src);
```

Table 19: DHCP code in udp_input (lwIP)

The code comments (not shown in Table 19) state that when ***LWIP_DHCP*** is enabled, packets to ***DHCP_CLIENT_PORT*** may only be processed by the DHCP module, no other UDP PCB may use the local UDP port ***DHCP_CLIENT_PORT***. Furthermore, all packets for ***DHCP_CLIENT_PORT*** not coming from ***DHCP_SERVER_PORT*** are dropped.
There is a separate DHCP protocol control block, with an associated receive function (via a function pointer). The associated function is normally ***dhcp_recv()***, defined in dhcp.c.

7.5.2 The PBMS BOOTP Server

A BOOTP server has been implemented in PBMS. Nera only required a placeholder for the actual processing of a BOOTP request and creation of a BOOTP reply. The implementation currently only receives the packet; the transmission of the reply is not implemented. This was only due to a lack of time, not any constraint in the design of PBMS. Following the approach of the DHCP client, ***struct netif*** now contains a BOOTP struct.

Have a look at Figure 14. A BOOTP message has been received from the network and lies in ***mbox.*** The Tcpip thread fetches this message and processes it like a normal packet. The fact that it is a BOOTP request is first detected in ***ip_input***, and the BOOTP specific receive function (described below) is called from ***udp_input***.

Initial BOOTP detection in ***ip_input***:

```
#if PBMS_BOOTP
  if (netif == NULL) {
    /* remote port is BOOTP client? */
    if (IPH_PROTO(iphdr) == IP_PROTO_UDP) {
      if (ntohs(((struct udp_hdr *)((u8_t *)iphdr + iphdr_hlen))->dest)
        == BOOTP_SERVER_PORT)
        {
        netif = inp;
        check_ip_src = 0;
      }
    }
  }
#endif /* PBMS_BOOTP */
```

Table 20: BOOTP code in ip_input() (PBMS)

The handling of BOOTP in ***udp_input*** is a simplified version of the DHCP code in Table 19.

```
#if PBMS_BOOTP
  /* if pcb isn't set to point to anything again within this block, the packet is
     dropped. */
  pcb = NULL;
  if (dest == BOOTP_SERVER_PORT) {
    if (src == BOOTP_CLIENT_PORT) {
      /* This test is mainly to see that PBMS has been properly initialised */
      /* Depends on short-circuit evaluation */
      if ((inp->bootp != NULL) && (inp->bootp->pcb != NULL)) {
        /* no address checks for now */
        pcb = inp->bootp->pcb;
      }
    }
  } else
#endif /* PBMS_BOOTP */
```

Table 21: BOOTP code in udp_input() (PBMS)

There is a separate BOOTP protocol control block in ***struct netif*** which points to a BOOTP receive function. The BOOTP struct and functions are defined in bootp.h and bootp.c. The reply will be sent by the BOOTP receive function:

Function: ***static void bootp_recv(…)***
File(s): bootp.c
Pseudo code for sending the BOOTP reply:

```
  struct pbuf *bootp_reply = mem_malloc(sizeof(struct pbuf));
  if( bootp_handler(p,bootp_reply) == ERR_OK ){
    udp_sendto(pcb?,bootp_reply,addr?,port?);
  }
```

7.6 BSD Socket interface

7.6.1 Nonblocking send

7.6.1.1 Nonblocking send in normal BSD sockets

According to [50] ***sendto*** (using a datagram socket) should return ***EWOULDBLOCK*** if there is no available space for the message to be transmitted. In [51], in relation to write sets for ***select***, it is stated that unless the low water mark for UDP is less than the send buffer size, a UDP socket is always writable. Chapter 16.1 [51] states that there is no actual UDP send buffer in the most common operating systems. UDP traffic can however be affected by buffering and flow control, so one must know the internals of the given system to be sure.

In lwIP there is in fact output buffering for UDP, as will be described here.

7.6.1.2 Nonblocking send in lwIP

The lwIP documentation and code comments clearly state that nonblocking send is unimplemented. Selected code comments in sockets.h, lwIP 1.3.0, also state the following related properties:

- Send and receive low water mark unimplemented
- Send timeout unimplemented

How nonblocking send is implemented in PBMS is described in chapter 6.7.1.1, see Figure 7 and Figure 8, as well as the paragraph right after the figure. How to use the nonblocking send and receive is described in the user documentation (chapter 3). This chapter only discusses some specific implementation issues.

7.6.1.3 Potential blocking points for *sendto*

One part of PBMS that queues outgoing UDP traffic is the collection of message queues that make up the priority queues. The first operation that might block in regular lwIP is ***sys_mbox_post,*** and this is an event that may occur frequently when transmitting high volumes of low priority traffic. After a message has been posted, there is also a potentially long wait for a semaphore to be signaled, but this only applies to low priority transmissions.

The implemented solution, to use ***sys_arch_mbox_trypost*** instead of ***sys_mbox_post***, is adequate as long as no new buffering is introduced in the lower layers of the target system port. Specifically this depends on that there is no potential blocking or queuing points once ***do_send*** is executed.

7.6.1.4 Configurable nonblocking send

There was some complexity related to making the nonblocking send configurable, so it is currently not. Nera specified that only nonblocking operations would be used. In the event that blocking send should be made available as well, the following issues must be considered.

How to determine dynamically that an operation should be nonblocking:

1. The variable ***flag*** in ***lwip_socket*** has the ***O_NONBLOCK*** option stored, if enabled.
2. The socket index is stored in the variable ***socket*** in the associated ***struct netconn***.
3. In ***tcpip_apimsg()*** all available data is in ***struct api_msg***.
4. That struct again contains a ***struct api_msg_msg***, which has a pointer to the ***netconn***.
5. Thus the socket number is available like this: ***apimsg->msg.conn->socket***.

So far the socket number has been determined. To access the ***flag*** variable one must use ***getsockopt,*** unless the existing interfaces are modified in some way. A call to ***getsockopt*** for every single attempted ***sendto*** seems like a bad idea, since it effectively doubles the amount of messages the Tcpip thread must handle for each execution of ***sendto***.

7.6.2 Nonblocking receive

According to the available documentation nonblocking receive is implemented. This has been verified on the test platform, and will be tested on the target system as well.

In ***lwip_recvfrom*** the precondition for returning ***EWOULDBLOCK*** is that either the socket flag is set to ***O_NONBLOCK***, or that the supplied flag for that single operation is ***MSG_DONTWAIT***. The actual test for whether a receive operation would block is: "!***sock->rcvevent***." For that expression to evaluate false, ***sock->rcvevent*** must be different from 0.
The variable ***u16_t rcvevent*** is part of ***struct lwip_socket***.

7.6.3 Select

The lwIP socket ***select*** function implements everything but the exception state. This makes it suitable for PBMS according to the design consideration in chapter 5.3.1. The mentioned overhead on the mapping from port to socket is still to be determined.

7.6.4 Setsockopt

Message requeuing due to altered priority is not implemented in PBMS.

7.7 The Integrity port of PBMS

The porting process is well documented at the lwIP ScribbleWiki [5]. There are three general tasks that will be done for the target system:

- Writing the OS interface
  - http://lwip.scribblewiki.com/Porting_for_an_OS
- Writing a device driver
  - http://lwip.scribblewiki.com/Writing_a_device_driver
- Configuring memory pools
  - http://lwip.scribblewiki.com/Custom_memory_pools

7.7.1 Memory Pools

This feature was not operational in the Linux port. Due to a lack of time this is postponed until the Integrity port has been written for PBMS. Once the port is functional the pools will be designed.

The requirement concerning the efficiency of the message buffers (PBMS1520) will be evaluated once the memory pools have been configured. Configuring pools of any desired size, as described in chapter 5.3.2, can be done by using the custom memory pools.

7.7.2 The Generic OS interface

In chapter 5.3.4 the desired degree of modularization is described. This is also part of the requirement specification, see chapter 4.2.4 on modifiability.

Since the message queue is provided with an abstract interface, it is loosely coupled to the scheduling logic. The scheduling logic itself is defined at one place, so this should also be interchangeable. The strict priority queuing is implemented by the for-loop in Table 11. The loop starts at the highest priority level every time the thread function executes, and always chooses the packet with the highest priority.

It is expected that much of the Linux port can be reused when coding the Integrity port. The message queues in the Linux port are for instance linked lists that only require the generic lwIP semaphore.

7.8 LwIP tasks, bugs and patches

The development of lwIP is an ongoing open source project, and a few of the currently open tasks, patches and bugs [4] are relevant for PBMS. Fixes to these may be useful to apply in PBMS as well.

7.8.1 Task #7865: Implement non-blocking SEND operation (socket)

There is a nonblocking send in PBMS, but it can be interesting to compare this with an implementation for the whole lwIP stack. There could for instance come a solution that allows run-time enabling or disabling of nonblocking send.

7.8.2 Patch #6483: Stats module improvement

The stats are usefull now, and increased configurability can make them even better.

7.8.3 Bug #23240: recv_udp increases counters for available receives before netbuf is actually posted

An unprotected shared counter is used to keep track of how many pending receives there are for a given receive message box. There are actually two counters, but the one at the socket level is properly protected by semaphores. The one in question is at the Netconn API level.

My proposed solution, as well as some suggestions from the community can be found at [4], bug #23240. What is currently implemented in PBMS is to increase the counters only if the ***netbuf*** is successfully posted. This solution has some potential flaws, as described in the follow ups to the bug report. As the comments suggest there are many possible interleavings to analyze, so I would favor an approach using an OS-level mutual exclusion mechanism.

The comments from the community also indicate that the generalized semaphore offered by lwIP is insufficient. For the implementation of PBMS it is not a problem to use a target-specific mechanism to resolve this bug, for instance a recursive mutex.

7.8.4 Bug #23408: Deadlock on sys_mbox_post sys_mbox_fetch

This is stated to be relevant for systems with small values of ***SYS_MBOX_SIZE*** (40 and less), which may be the case for PBMS. There seems to be a suggested patch that solves the problem, but the bug report has not been closed yet.

7.8.5 Bug #21433: Calling mem_free/pbuf_free from interrupt context isn't safe

If the network interface on the target system is to be implemented using interrupts, this bug and the discussion surrounding the proposed solutions should be taken into consideration.

8 Test Plan

8.1 Introduction

Verification: To establish the truth of correspondence between a software product and its specification (from the Latin veritas, “truth”).
Validation: To establish the fitness or worth of a software product for its operational mission (from the Latin valere, “to be worth”).

Verification*: Are we building the product right?*

Validation*: Are we building the right product?*

Table 22: Definitions of verification and validation from [52]

A sizable part of PBMS has been verified according to the requirements stated in chapter 4 on the Linux test platform. After the Integrity port for PBMS has been developed, PBMS will be validated for use in the target system. This test plan will therefore describe both the Linux platform tests and the test methods to use in the target system validation.

8.2 Linux platform test overview

Each successive step of these tests uses code that was tested in the previous step, so that the tests gradually lead up to a fairly complete system test of PBMS.

8.2.1 Linux platform overview

Hardware	Intel Pentium 4 CPU 2.00 GHz, RAM: 503,8 MB
OS	Ubuntu 7.10 (gutsy gibbon)
Kernel	Linux 2.6.22-14-generic
Compiler	gcc version 4.1.3 20070929 (Ubuntu 4.1.2-16ubuntu2)
Additional tools	Wireshark version 0.99.6

Table 23: Linux platform configuration

8.2.2 General settings

- ANSI compile
- Test disabled IP and UDP checksums using Wireshark
- Disable all unnecessary protocol support

8.2.3 Using loopback interface

- BSD Socket interface
- Priority scheme
- Nonblocking I/O
- Message queue error handling

8.2.4 Using TAP interface

- Socket-based communication between Linux and PBMS
    - IPv4 and UDP interoperability test, Wireshark to check packet formats
    - ARP test, Wireshark
    - Run without built-in debugs in PBMS to verify correct operation
- Test of deterministic execution time for internal send operation in PBMS

8.2.5 Performance measurement on the Linux platform

The performance of PBMS is measured via the time stamp counter of the Pentium 4 processor. Volume 2B of the *Intel 64 and IA-32 architectures Software developer's manual*, available at [53], describes how the counter is to be used. A set of questions concerning the "read time stamp counter" (RDTSC) operation answered at the official Intel software community [54] have also been used as a guide. According to those answers:

"Measurements with RDTSC are most credible if the number of clocks between the pair of RDTSC instructions is at least a few thousand, preferably tens of thousands."

According to [53] the RDTSC operation loads the current value of the processors time stamp counter into the EDX:EAX registers. The measurement method is then basically to read this value before some operation, read it afterwards, and compute the difference. Before reading the value a serializing instruction is executed.

```
void pbms_perf_start_serialized(){
  __asm__ __volatile__ (
     "xorl %%eax,%%eax \n        cpuid"
     ::: "%rax", "%rbx", "%rcx", "%rdx");

  __asm__(".byte 0x0f, 0x31" : "=a" (__c1l), "=d" (__c1h));
}

void pbms_perf_stop_serialized(){
  __asm__ __volatile__ (
     "xorl %%eax,%%eax \n        cpuid"
     ::: "%rax", "%rbx", "%rcx", "%rdx");

  __asm__(".byte 0x0f, 0x31" : "=a" (__c2l), "=d" (__c2h));
  perf_print_file(__c1l, __c1h, __c2l, __c2h, f3);
}
```

Table 24: C code for RDTSC operation

According to the Intel 64 and IA-32 architectures Optimization reference manual [53] *"(...) the RDTSC instruction is not serializing or ordered with other instructions. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDTSC instruction operation is performed."* Serialization is therefore done for increased accuracy in the measurement code.

Limitations

The time measurement shown in Table 24 uses two pairs of static variables, which requires a certain usage pattern to be reliable. This is discussed in the test report, see chapter 9.

8.3 Integrity platform test overview

8.3.1 Target system IP Performance Measurements

According to requirement PBMS1410 (in chapter 4.2.3) the round-trip time (RTT) should be measured. It is a defined goal to avoid the problems related to introducing duplicates upon retransmission (see chapter 1.3.1). For increased accuracy one could also measure the one way delay from the supervisory unit (SU) to a card (see Figure 2: Target system network topology) and likewise for the response message. Then one could compute how much of the measured RTT which is caused only by the latency in the card.

One important argument for using both metrics is given in [55]. In QoS-enabled networks there may be differences in how the traffic is prioritized through two different paths in the network. PBMS will depend on that the cards set the TOS field properly in all response packets, and that the switches schedule the packets strictly. In theory a packet should then be treated equally in both directions, but that must be verified through testing. It is therefore important to use both one-way and round-trip measurements.

Requirement PBMS1400 states that the intent of PBMS is to achieve a deterministic round trip time. There may be a variable latency in some of the cards in the target network. An important goal is to know precisely when a packet is lost, as opposed to delayed. By using both one-way and round-trip measurements, the latency in the cards can be estimated.

The “Framework for IP performance metrics” (IPPM) [56] published by IETF defines many of the terms used in the one-way and round-trip metrics. The Type-P in these experiments will be defined as UDP/IP protocol packets, of variable size but less than or equal to the Ethernet MTU. The IPv4 precedence bits will be different from 0 in some of the packets.

8.3.2 A Round-trip Delay Metric

RFC2681 [55] describes a round trip delay metric for IPPM. There are three elements in this measurement metric:

1. A Singleton, Type-P-round-trip-delay.
   This is defined as the time from the first bit of the Type-P packet is on the wire, until the last bit of the reply has been received (at the sender side).
2. Sample, Type-P- round-trip-delay Poisson stream.
   Several singleton transmissions will be carried out, and a sample set will be drawn from these.
3. Several statistics can be developed based the measured samples.

In point 2 Poisson sampling is generally recommended since it minimizes any perturbation of the measured sizes, and enables detection of periodic behavior.

8.3.3 A One-way Delay Metric

RFC2679 [57] describes the one-way metric. It is quite similar to the round-trip delay. It is specifically stated in [55] that a round-trip delay does not equal the two corresponding one-way delays, as these do not take into consideration processing time at the receiver side.

The three elements according to [55]:

1. Singleton, Type-P-one-way-delay.
   The time from the first byte is on the wire until the last byte is received.

2. Sample, Type-P-one-way-delay Poisson stream.
   Similar to round-trip.
3. Statistics

An outline of the methodology for exact measurements:

- Synchronized clocks
- Randomized padding
- Source host timestamps packet
- Destination host does a timestamp immediately after the reception is complete

8.3.4 General settings

A compilation for the ARM architecture to measure the code size is to be carried out.
The memory pools are to be configured, and measurements to find a suitable range of available sizes can be done.

8.3.5 Interfaces

The socket API is to be tested for user errors.

8.3.6 Protocols

Upon completion the BOOTP protocol must be tested.

8.4 Potential delays in the strict priority scheme

Even for the highest priority level defined in PBMS there are some potential internal processing delays.

8.4.1 Data aggregation

If several data streams are aggregated at the same priority level, there will be a worst-case where all streams send and/or receive packets simultaneously.

8.4.2 Tcpip thread latency

When examining the Tcpip thread in isolation the worst-case latency for a single top priority packet is the time it takes to execute one whole iteration of the for-loop in the function ***tcpip_thread***. The packet can arrive at the top level just when the thread examines the second-highest priority level. By doing measurements this specific delay time can be estimated. However, PBMS consists of more than one thread; the Linux port has at least three.

8.4.3 Netif and application thread(s) interference

The network interface thread and the Tcpip thread have only been tested at equal priorities. It is then possible that the Tcpip thread is scheduled out in the middle of a priority operation to allow the Netif thread to run. The same also applies to other thread with equal or higher priority (e.g. application threads). One could try to give priority to the Tcpip thread to achieve increased determinism. This will at the same time give decreased priority to the incoming packets (Netif thread), which could lead to more full buffers and dropped packets.

8.4.4 OS interference

Depending on the assigned OS level priorities in the target system, some latency due to the applied scheduling policy is to be expected. This comes in addition to the specific latency in the Tcpip thread, and the potential delays due to traffic aggregation.

8.5 Architectural support for testing in lwIP and PBMS

8.5.1 General categories

Architectural testability tactics can be sorted in two main categories according to Bass et al. [1]:

- Manage I/O
  - Record/playback
  - Separate interface from implementation
  - Specialized access routines/interfaces
- Internal monitoring
  - Built-in monitors

What lwIP offers is:

1. Platform generic debug printouts
2. Collection of statistics
3. Platform specific performance measurement module with a uniform interface

These are all special cases of the general architectural tactic *Built-in monitors*. There is no architectural support for any of the *Manage I/O* tactics mentioned in [1]. Managed I/O can be very useful during testing, as it is sometimes difficult to trigger specific error conditions. Such functionality has been added in a few places in PBMS. The purpose was to trigger certain error conditions related to code implemented in PBMS, not lwIP in general.

8.5.2 Implemented test functionality in PBMS

In the message queue module I added a count method, so that I could check the status of each priority queue serviced by the Tcpip thread. The implementation was a simple linear search, as shown in Table 25.

```
u32_t
sys_arch_mbox_count(struct sys_mbox *mbox){
  u32_t count;
  int pointer;

  count = 0;
  pointer = mbox->first;

  sys_arch_sem_wait(mbox->mutex, 0);

  if (mbox->first == mbox->last) {
    sys_sem_signal(mbox->mutex);
    return 0;
  }

  while(pointer!=mbox->last){
    count++;
    pointer++;
  }

  sys_sem_signal(mbox->mutex);

  return count;
}
```

Table 25: Count method for message queue module (sys_arch.c)

The count method in Table 25 can for instance be used as in the code segment of Table 26.

```
#if TOS_MBOX_COUNT
print = 0;
for(debug_index=0; debug_index<PBMS_PRIORITY_LEVELS; debug_index++ ){
  values[debug_index] = (int) sys_arch_mbox_count(mboxes[debug_index]);
  if( values[debug_index] > 0 ){
    print++;
  }
}
if(print > 1){
  LWIP_DEBUGF(TOS_MBOX_COUNT, ("TCPIP Thread mbox status: low:[%d] norm:[%d]
    high:[%d]\n", values[0], values[1], values[2]) );
}
#endif
```

Table 26: Count method used in tcpip.c

To force some error conditions in relation to the message queues I added the following code to ***tcpip_apimsg()***, the function which among other things is used to send messages. This was used to test the correct behavior of the nonblocking send, as it was difficult to fill the message queues using the normal send operation on the Linux test platform.

```
#if DEBUG_PRI_Q
if(tos == IPTOS_PREC_ROUTINE){
  input_queue = PBMS_OUTPUT_PREC_ROUTINE;
  do {
    i = sys_mbox_trypost(mboxes[input_queue], &msg);
  } while( i == ERR_OK );
}
#endif
```

Table 27: Trigger full message queues in tcpip_apimsg(), tcpip.c

9 Test Report

9.1 Introduction

The tests have been carried out according to the outline in chapter 8.2. The single most important result is from the test for determinism in the message scheduling of PBMS. There is no reason to believe that the TAP interface can emulate the target system network; what has been measured is therefore the internal processing time of a single message transmission. The one-way and round-trip tests introduced in chapter 8.3 will only be carried out on the embedded target.

Much of this chapter is devoted to presenting the performance measurements. LwIP has a fairly large user community, and reported bugs are resolved quickly. Therefore much of the existing lwIP code has been tested rather briefly. Most of the available time was devoted to testing features that were implemented in PBMS.

The TSC measurements were also carried out while mapping the high priority traffic to the same queue as the low priority bulk transfers.

9.2 The Linux TAP interface

In the performance tests network communication was emulated using the TAP interface of the Linux kernel. The TUN/TAP provides packet reception and transmission for user space programs [58]. It can be seen as a virtual Ethernet device. Instead of receiving packets from the physical medium, TAP receives them from user a space program. Instead of sending packets via the physical medium it writes them to a user space program. TAP works with Ethernet frames and TUN works with IP packets. To also test some ARP functionality I chose TAP for the Linux platform tests.

9.3 General settings and sockets

The tasks described in chapter 8.2.2 offered no problems. The next step was to test the socket interface via the lwIP loopback interface. I had one lwIP socket communicate with another, running two separate threads in the test program. The test programs from various stages of development are server.c and test1.c - test4.c. All test programs were archived along with the Makefile used to compile them and the lwipopts.h configuration they were run with.

The normative standard describing the sockets interface is The Open Group Base Specifications Issue 6 [7]. The socket interface was used for all testing of PBMS, but few error cases have been deliberately triggered. The proper implementation of nonblocking receive was tested in some detail.

9.3.1 Nonblocking socket I/O

LwIP blocks if the nonblocking mode is not enabled, and returns ***EWOULDBLOCK*** when appropriate. In lwIP the numerical value of ***EAGAIN*** equals ***EWOULDBLOCK***. Using the flag ***MSG_DONTWAIT*** for a single receive is supported, as well as using ***lwip_ioctl*** to make all receives nonblocking. By using the code in Table 27 the nonblocking send was tested as well. This worked according to the description in chapter 7.6.1.

9.3.2 Fixed bug in Linux port

The message boxes were specifically tested for the case when they are filled up, in relation with testing the nonblocking send. During that process I found a bug in the Linux port and posted a proposed fix for it. This was accepted and will be part of lwIP 1.3.1. The description of this on Savannah must be accessed via a direct link[5] since the bug report is now closed.

The problem was that the function ***sys_mbox_trypost***() in sys_arch.c did not release a certain semaphore upon an unsuccessful read attempt. Once this problem was fixed the situation described in chapter 7.8.3 became evident. I implemented my temporary solution, which worked out fine in the tests. A safer solution, probably using a recursive mutex, will be implemented on the target platform.

9.3.3 Message Queues (*sys_mbox*)

Data transmission with a very small message box size was performed, and clearly showed the implemented packet drop scheduling; keep old packets and drop new arrivals. For package reception to always fail on full a message queue, not lack of ***NETBUF***s (i.e. memory), the defined size of ***MEMP_NUM_NETBUF*** must be larger than the number of receive message queues multiplied with the defined capacity for a single message queue, (***SYS_MBOX_SIZE***) plus one for attempted receives. The number of receive message queues equals the number of UDP ***netconn*** structs in PBMS.

In a constrained-resource scenario it might be preferable to fail on a full message box, rather than lack of memory. The message box implementation will be coded specifically for Integrity anyway, and it is then easier to include safety measures there. If it is possible to use the memory pool implementation directly without modifications then it is much easier to apply any patches, bug fixes and new features that might come out of the lwIP community.

[5] http://savannah.nongnu.org/bugs/?23230

9.4 PBMS Performance Tests

The results from three different test runs will be described and compared here. The first will be referred to as the **TSC basic test**, and uses the time stamp counter for measurements. The second is a variation of the basic test where the request-response thread has same stack priority as the bulk transfer threads. This will be referred to as the **TSC no priority test**. The third test run is the same as the basic, but with time measurement done via ***gettimeofday*** and Wireshark. This will be referred to as the **Wireshark basic test**.

A program using the regular Linux sockets was the receiving end in these tests.

9.4.1 Test scenario

File(s):

- request-response2.c
  For both tests using the time stamp counter (TSC)
- request-response1.c
  For the test using Wireshark and ***gettimeofday***

Scenario

- 10 threads, 20 sockets / thread → 400 PBMS sockets.
- All transmitted packets are 1446 bytes
- 1 high priority socket doing request-response
- Bulk transfer on low priority
- No OS level prioritization between the threads, only different TOS levels.
- Measure the delay from the beginning of ***lwip_sendto***(), see Table 28
  - To the end of ***ip_output_if()***, see Table 29 (TSC measurements)
  - Or until Wireshark detects the packet

```
  /* PBMS perf measurement, first socket allocated is priority */
  if(s==0){
    pbms_perf_start_serialized();
  }
```

Table 28: RDTSC added at the beginning of lwip_sendto

```
  if( ( ntohs(iphdr->_v_hl_tos) & 0xe0 ) == 0x20  ){
    pbms_perf_stop_serialized();
  }

  return netif->output(netif, p, dest);
}
```

Table 29: RDTSC added at the end of ip_output_if

9.4.2 Test limitations

The time measurement shown in Table 24 uses a single pair of static variables, which is unsafe for everything but a scenario where a single thread transmits one message and it is ensured that no further transmits are attempted before the Tcpip thread has finalized the send. Technically it is safe to call ***lwip_sendto*** again after ***ip_output_if()*** has done the time stamp, but there is no way to determine when that specific event has occurred at the sockets interface.

The test setup demonstrates a safe use, as another transmit is not attempted before a reply to the previous one has been received, although this depends on the behavior of the receiver side.

9.4.3 Test Code

```
static void socket_thread(void *arg){
  (declare variables)
  (most error handling removed throughout)

  /* SOCKET(), BIND() */
  for(i=0; i<SOCKETS_PER_THREAD; i++){
    ta->sockets[i] = lwip_socket(…);
    my_addr.sin_port = htons(ta->send_ports[i]);
    lwip_bind(…)
  }

  /* Signal main thread that init is complete */
  sys_sem_signal(ta->sem);
  sys_sem_wait(ta->sem);

  for(i=0; i<SOCKETS_PER_THREAD; i++){
    remote_addr.sin_port = htons(ta->recv_ports[i]);
    for(j = 0; j < ta->packets ; j++ ){
      big_send.send_value = j;
      send_count = lwip_sendto(…);

      if( send_count == -EWOULDBLOCK){
        printf("%s: Socket%d EWOULDBLOCK\n", name, ta->sockets[i]);
        break;
      }

    }
  }
}
```

Table 30: Routine priority thread, bulk transmit

The Linux program which interacts with these threads does a selective receive on a known range of ports, and transmits a reply with a predefined TOS when a packet is received from the priority socket.

The main function creates all threads, and then signals for them to begin transmission once all threads are ready. The **TSC basic test** runs 10 bulk transfer ***IPTOS_PREC_ROUTINE*** priority threads, and one request-response running at ***IPTOS_PREC_PRIORITY***. The **TSC no priority test** runs all threads at ***IPTOS_PREC_ROUTINE***. In both tests all other stack-internal processing also runs at ***IPTOS_PREC_ROUTINE***.

Table 31 shows the main function which starts all the bulk transfer threads and the priority thread. Note that the priority thread starts transmission after all others have been started. Table 32 is an overview of the priority thread code.

```
/* Thread initialization */
sys_thread_new("priority_thread", ...);
printf("%s: Main: Waiting for priority thread init.\n", name);
sys_sem_wait(pri_arg.sem);

printf("%s: Main: Waiting for thread inits:", name);
for(i=0; i<NUM_THREADS ; i++ ){
  sys_thread_new("socket_thread",...);
  printf("%d ",i);
  sys_sem_wait(threads[i].sem);
}
printf(" :: All done.\n");

/* TX START SYNCHRONIZATION */
printf("%s: Main: Routine threads begin TX: ", name);
for(i=0; i<NUM_THREADS ; i++ ){
  printf("%d ",i);
  sys_sem_signal(threads[i].sem);
}
printf(" :: All started.\n");

printf("%s: Main: Signalling priority thread to begin transmission.\n", name);
sys_sem_signal(pri_arg.sem);
```

Table 31: Segment of main() in PBMS test

```
static void priority_thread(void *arg) {
  (declare variables)
  (most error handling removed throughout)

  /* SOCKET(), BIND() */
  ta->socket = lwip_socket(…);
  lwip_bind(…);

  /* SETSOCKOPT() - TOS */
  lwip_setsockopt(ta->socket, IPPROTO_IP, IP_TOS, &(ta->send_tos), …);

  /* Signal main thread that init is complete */
  sys_sem_signal(ta->sem);
  sys_sem_wait(ta->sem);

  /* Blocking receive here */
  /*lwip_ioctl(ta->socket,FIONBIO,&ioctl_arg);      */

  for(i = 0; i < ta->packets ; i++ ){
    big_send.send_value = i;
    send_count = lwip_sendto(…);

    if( send_count == -EWOULDBLOCK){
      printf("%s: Socket%d EWOULDBLOCK, breaking send loop\n", name, ta->socket);
      break;
    }
    else if(send_count == -1){
      /* Error handling */
    }
    else {
      recv_count = lwip_recvfrom(…);

      if( recv_count <= 0 ){
       /* Error handling */
      }
      else {
        rx++;
      }
    }
  }
}
```

Table 32: Priority thread, request-response

9.4.4 The calculated execution times

The processor is 2 GHz (2e9 clock cycles / second), and the measured values are the number of processor cycles elapsed. To calculate the number of elapsed microseconds the following formula is then used:

Elapsed time (μs) = Elapsed cycles / Processor speed = **Elapsed cycles / 2000** (μs)

Hibernating / energy saving operating systems will slow down the processor, thus making any calculations on the number of ticks unreliable. The results are exact when it comes to how many processor cycles that elapsed between the measurement points in Table 28 and Table 29. What could be wrong is the conversion to microseconds, as a lower clock frequency implies that the operations in fact take more time.

The TSC measurements will be compared with the Wireshark measurements to evaluate the reliability of both measurement series. Since the main goal is to show determinism, not a certain execution speed, I do not consider the issue of potentially lower execution speed to be a major problem. It would have been possible to measure the actual speed of the processor during the experiments as well. In the end the program will be compiled for another hardware architecture, with a different compiler.

9.4.5 The TSC basic measurement result

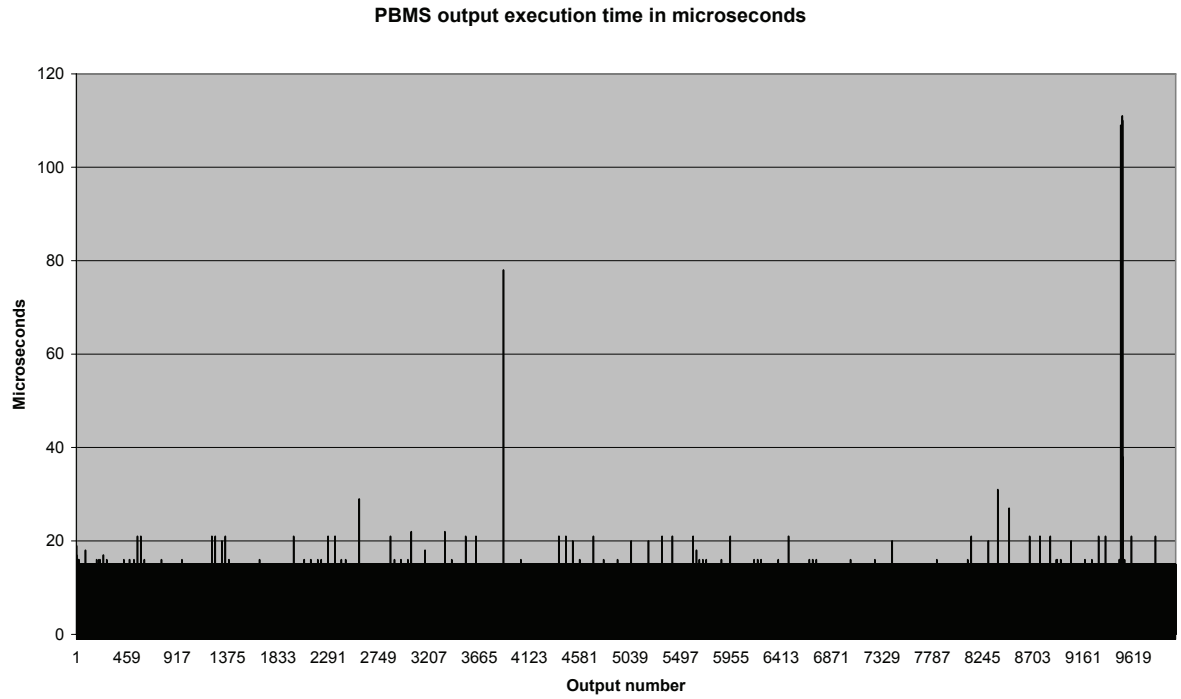

Figure 15: TSC measurement with prioritization

The maximum execution time measured was 111 microseconds; the minimum was 15 microseconds. 9889 out of 10000 samples were of the minimum value. According to the code segment in Table 31 the priority thread is started last, yet it always finished first in this test scenario.

Possible design-based explanations for the deviating values are discussed in chapter 8.4.

A calculated delay of 15 microseconds equals roughly 30400 clock cycles. According to the forum posts made by Intel engineers (see chapter 8.2.5) these timing measurements can therefore be considered reliable. Again, the conversion to an actual elapsed time interval is not important in evaluating these results. What is important is the number of elapsed cycles. My results are therefore not dependent on the statements made by the Intel engineers.

No OS level prioritization was used during this tests. I did try to run the test with the maximum ***nice*** values (-20) on the test processes, but this did not improve the results.

9.4.6 The TSC no priority test result

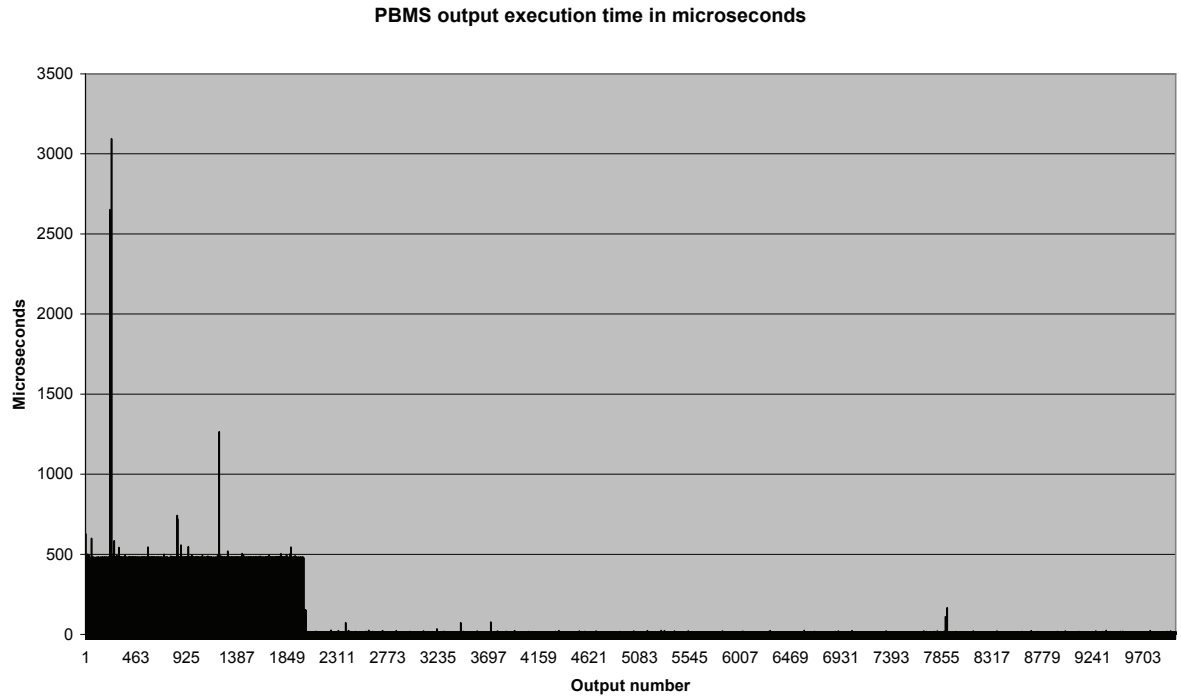

Figure 16: TSC measurements without prioritization

The ”priority” thread (which is not prioritized here) finished last. There is a marked decrease of the delay after 2000 samples. That is when all the bulk transfers have finished. The first 2000 samples show a delay of about 480 microseconds. The 8000 remaining samples have a delay of 16 microseconds.

9.4.7 The Wireshark basic test result

Measured size	**Value**
Min	21 microseconds
Max	1550 microseconds
Average	22,16 microseconds
Greater than 100 microseconds	9 Samples
Greater than 200 microseconds	1 Sample
Distribution:	
21 microseconds	3196 Samples
22 microseconds	6493 Samples
23 microseconds	148 Samples
>23 microseconds	163 Samples
Sum:	10000 Samples

Table 33: Wireshark basic test results

These values were calculated by printing each ***gettimeofday*** timestamp to a file, and then copy these and the values from Wireshark into a spreadsheet. ***Gettimeofday*** would be performed at the same spot where ***pbms_perf_start_serialized()*** is done in Table 28.

Initially the file buffer was flushed for each measured value, but the values were markedly improved when i removed the buffer flushing. It will however be flushed at some points, so that may explain some of the variations.

These measurements do not have the same potential sources of errors as the TSC measurements. The measured time interval is however not the same, so they are not directly comparable.

9.4.8 A Priority scheme for the Linux test

To ensure that no other OS processes interfere with the operation of PBMS during testing, one can try to schedule these threads with ***SCHED_FIFO*** instead of the normal O(1)-Linux scheduler. A priority scheme for all threads involved will then have to be worked out.

The following threads would run at ***SCHED_FIFO:***

- Tcpip thread
  - This one is continuously polling, insert one yield per iteration.
- TAP interface thread
  - This one is using select, no problem.
- ***priority_thread*** (sender)
- 10 bulk transfer threads
- Main thread
- Receiver program

The default in PBMS is to have all stack threads run at equal priorities, with no specific scheduling algorithm. It seems difficult to ensure that a test using SCHED_FIFO will behave in the same manner. If one could prove determinism given a certain scheduling strategy, the target system would have to implement the scheduling strategy as well, not just use PBMS.

A technical issue is that the lwIP thread priority argument is unused. The generic OS interface would then have to be modified, or the test could be done by creating the Posix threads directly. Using Posix threads directly would complicate matters further, as these are not available in the target system.

It is also uncertain whether running PBMS at ***SCHED_FIFO*** should give any significantly improved results at all. The **TSC basic test** was run with a ***nice*** value of -20 set for both processes, but that did not seem to improve the measured values. This could indicate that the variations stem from the Tcpip thread latency, and TAP interface and application thread interference, as discussed in chapter 8.4.

10 Results

Two categories of results will be presented, and then discussed in the next chapter.

- The implemented functionality in PBMS
- The quality of the PBMS implementation

The performance measurement results are already presented in the test report, so these will not be described further in this chapter.

The requirements could be stated with three degrees of necessity; essential, conditional and optional. No optional requirements were documented. The design goal of the entire stack was to offer a deterministic transfer time, see requirements PBMS1100 and PBMS1400. Those will be regarded as more important than any other essential requirement from the requirement specification.

The process requirements were all followed, as lwIP has a BSD-type license, and all code in PBMS is written in ANSI C.

10.1 The required PBMS functionality

Whether a requirement has been reached will be indicated with a simple yes or no. The answers will be justified in the text following Table 34. Some requirements are classified as collections. Whether these have been reached will be indicated with a rough percentage, and will the discussed further afterwards.

The purpose of the rightmost field in the table is to state if the requirement was reached simply by using lwIP alone, or if additional work done.

Requirement	Classification	Reached?	How?
PBMS1000	Collection	75% (the whole project)	COTS and implementation
PBMS1100	Essential	Yes	COTS and implementation
PBMS1110	Conditional	Yes	COTS and implementation
PBMS1200	Collection	90%	COTS and implementation
PBMS1210	Essential	Yes	COTS
PBMS1220	Essential	No	Implementation
PBMS1230	Essential	Yes	COTS
PBMS1240	Essential	Yes	COTS

Table 34: PBMS Functional requirements coverage

The completed percentage stated for the entire project is just a rough figure which is elaborated in the concluding chapter of this report.

BOOTP is not finished, but there is no fundamental design condition which makes it difficult to complete, so the general 90 % rating should be reasonable. The ARP, UDP and IPv4 COTS modules have been used widely by others, but they must still be validated in the target system before 100% completion can be claimed.

10.2 The required PBMS quality attributes

These requirements will be presented in the same manner as the functional ones. The target system integration is to be carried out after the completion of this project. Many requirements are therefore strictly not reached, as they have not been validated on the target system yet.

Requirements PBMS1600 and PBMS1610 are specific to the target integration process, and are therefore not evaluated together with the others. PBMS1600 is concerned with the "upwards" interface of PBMS. Since a BSD socket interface is implemented PBMS should be suitable within the existing architecture, as shown in Figure 5.

The evaluation of PBMS1610 is concerned with the "downwards" interface of PBMS. Nera's general consideration is that it is feasible to implement the generic network interface on top of the existing Ethernet drivers. A detailed discussion of that interface can not be published in this report.

Requirement	Classification	Reached?	How?
PBMS1300	Conditional	Yes	COTS and implementation
PBMS1400	Essential	No	COTS and implementation
PBMS1410	Conditional	No	COTS and implementation
PBMS1500	Collection	?	COTS and implementation
PBMS1510	Conditional	No	COTS and implementation
PBMS1520	Conditional	No	COTS and implementation

10.2.1 The performance requirements

The desktop tests have focused on the stack-internal delay. RTT measurements have not been performed yet. The stated requirements PBMS1400 and PBMS1410 can therefore not be considered reached. This is discussed further in the next chapter.

10.2.2 Memory footprint

According to the lwIP documentation the stack only requires "*40 kilobytes of ROM*", but that claim can not be corroborated by any reliable source. PBMS has not been compiled for ARM yet, so the requirement is not properly verified. The efficiency of the message queue implementation has not been verified either. The custom memory pool features of lwIP / PBMS should however enable a suitable solution according to PBMS1520 as well.

11 Discussion and related work

The single most important quality aspect of PBMS was to offer a deterministic transfer time. The first subchapter summarizes the design rationale given in chapter 6.4 and the performance measurements from the test report. The use of COTS will also be discussed, and some related work is presented at the end of this chapter.

11.1 Deterministic transfer time

11.1.1 Enabling technologies

One key precondition for a deterministic transfer time over Ethernet is that the network is fully switched, since the stochastic CSMA/CD protocol is no longer required [3]. The fact that the switches were QOS-enabled was also essential in the PBMS design. These two features will in theory ensure the proper behavior once a packet enters the network. This will of course have to be validated through system tests.

The decision to use a per-packet priority scheme to ensure low delay is supported by Schantz et al. [28]. Kos et al. [41] simulated priority queuing and found it to be suitable for providing low delay. Schmitt and Zdarsky [42] argue that a strict priority scheme is a simple and available QOS solution, which has been confirmed in this project. The technology is indeed available, and the algorithm is easy to implement.

Ferrari et al. [43] did a measurement based analysis of the priority queuing algorithm, and found that it was one of the most effective algorithms to minimize queuing delay. Their study also showed that a strict priority scheme will be sensitive to First-Come-First-Serve (FCFS) scheduling in any network element, for instance the transmitting node.

11.1.2 Prioritization in all network elements

It is known that the target system has switches that can prioritize strictly. A precondition for a deterministic RTT would, according to the results of Ferrari et al., be that all other network elements could prioritize strictly as well. Through the use of PBMS the supervisory unit will prioritize incoming and outgoing data streams strictly. This has feature been implemented and desktop tested. The target system performance will depend on the assigned OS level task priorities, but the fundamental mechanism for strict prioritization is available.

11.1.3 Measured performance

When the traffic is prioritized strictly the tests in chapter 9 gave an execution time of 15 microseconds in 99% of the transmissions. When prioritization was disabled the execution time was about 480 microseconds when there were competing transmissions. The time stamp counter (TSC) measurements had associated uncertainties, but the complementary measurements from Wireshark gave similar results.

The measurements were done with two time stamps. Both methods do the first time stamp in the socket ***sendto*** function. The TSC method does the second timestamp at the end of ***ip_output_if***, while the Wireshark measurements has the second measurement from Wireshark (hence the name). It is therefore to be expected that the Wireshark times are greater than the TSC times.

The TSC measurements of elapsed time are susceptible to the processor running at a lower speed than the stated 2 GHz. The Wireshark series rely on that the function ***gettimeofday*** and the program Wireshark use a common clock. A basic indication of reasonableness is that the Wireshark values are generally 5 microseconds larger than the TSC values. A better tactic could have been to run a test series using ***gettimeofday*** in ***ip_output_if*** as well.

The main goal is to show determinism, and the measurement series show some variations that must be explained. Regardless of the processor speed, the elapsed number of processor cycles is an exact figure one can reason about. The data material for the graph in Figure 15 is shown in Figure 17 in the original form; the number of elapsed cycles.

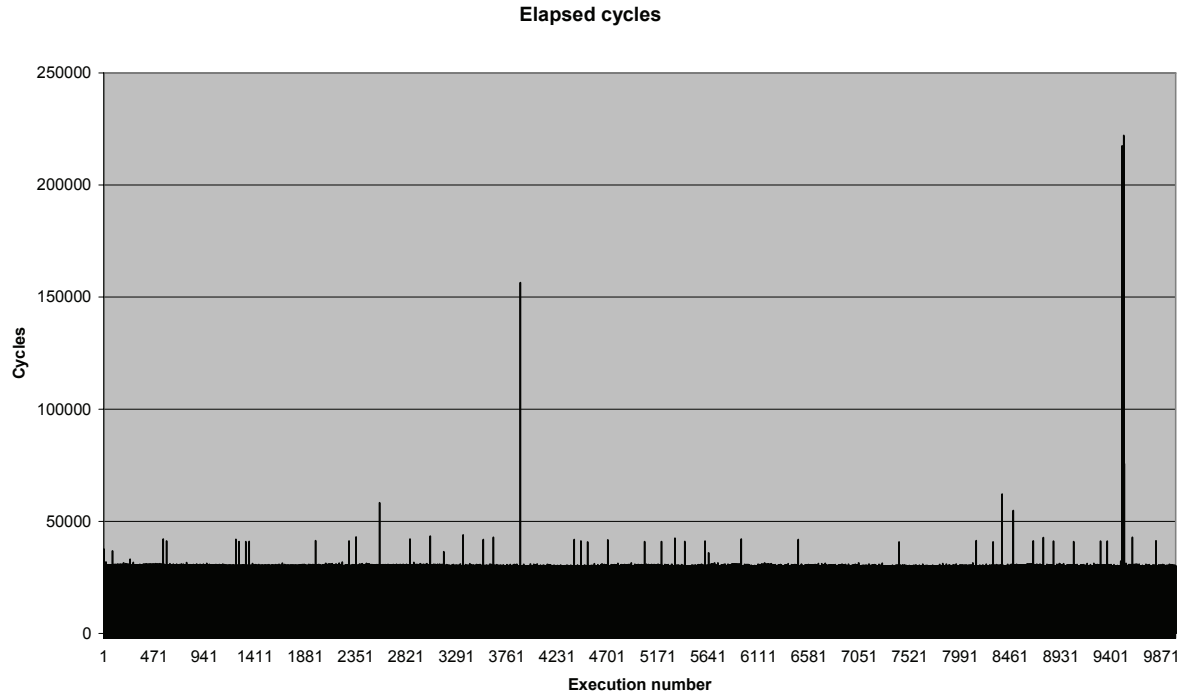

Figure 17: Elapsed execution cycles per socket send

The possible sources of errors discussed so far are data aggregation, Tcpip thread latency, Netif and application thread interference and OS interference. Data aggregation was by design not a problem in the test series, as there was only a single socket using the elevated priority class. I have no good basis for evaluating which of the others that caused the variations in Figure 17.

11.1.4 Target system validation

Based on the test experiences from the Linux platform, it is important to determine if a strict OS priority scheme involving PBMS should be worked out. If a definite worst-case time is to be found, the OS scheduling strategy will be a key factor. The results will also only be valid when that specific scheduling strategy is followed. Furthermore, if one wishes to reduce this worst-case time then a scheme for the stack-internal thread prioritization should be designed, and data aggregation should be avoided.

If the ultimate goal is to never introduce a single duplicate, then one must follow the worst-case execution strategy to the end, and use worst-case figures for all possible delays. If a requirement on the form "*The RTT should be lower than X time units 99% of the time*" is acceptable, then statistical measurements along with system tuning might give satisfactory results. An exact worst-case analysis can give a figure which is too large to have any practical value. A certain general throughput is also a requirement in this system, so a scheme that allows for some extreme values is perhaps better.

The implemented priority queuing gives a distinct reduction in transmission delay time. What has yet to be done is to validate that this stack-internal prioritization, combined with the target network technology, in fact ensures a deterministic transfer time. That will be done when PBMS is ported to the target system, after the completion of this thesis. The simulations done by Kos et al. [41] indicate that a strict priority scheme should not degrade the general performance for low priority traffic, but this will have to be verified in the target system.

The remaining network elements, the cards, must also be considered. Some of these may schedule traffic in a FCFS manner, and that would affect the RTT. To detect such conditions I have planned to test both one-way delays and round-trip delays on the embedded target. This will be done according to the standards published by the IP performance metric working group of IETF.

11.2 The use of PBMS in an embedded system

Some documented architectural risks in PBMS are described in chapter 6.5. These are examples of a general issue when using COTS. The desired functionality is probably available, but one has little control over the quality attributes [1]. One solution I propose is to rebuild the stack from smaller components. In doing so, it might become harder to incorporate bug fixes and updates from the lwIP community, so this is an architectural tradeoff point.

For embedded systems an important general quality attribute is availability, maybe up to twenty or thirty years. While lwIP seems to have a good track record, the continuous process of bug fixing and adding new features might not be compatible with the requirements of this target system. It may be a good idea to build PBMS from one clearly defined version of lwIP, and then only implement selected improvements in a controlled manner.

PBMS is already a customized version of lwIP, so the next logical step could be to remove all unwanted functionality. There is a significant amount of code in lwIP that is not used in PBMS, and this might decrease the modifiability and related quality attributes. A rough estimate is that PBMS only requires half of the lwIP code base, and perhaps as little as a third. Even though LOC may not be the best metric, maintaining about ten thousand code lines seems easier than building PBMS from a code base of thirty four thousand LOC.

The other possible approach is to maintain PBMS as a ***unified diff***, i.e. only keep track of the differences between lwIP and PBMS. Then one could “apply” PBMS to any new version of lwIP, to make use of added functionality, fixed bugs and more. However, PBMS is developed only for UDP, IPv4 and ARP. It is therefore not certain that a PBMS ***unified diff*** would be compatible with a future version of lwIP.

Since lwIP is an open source project there is no guarantee that interfaces will stay the same. There is for instance an ongoing discussion in the lwIP community of whether to use a stack level semaphore instead of the Tcpip thread, which would require a fundamental redesign of PBMS.

11.3 Project plan

The initial project plan can be seen in Table 1, see chapter 2.1. The actual progression of the individual tasks was as shown in Table 35.

Task																				
COTS Analysis								x												
Problem definition																				
Requirement Specification					rs1				*							rsf				
Test Plan									*											
Design Documentation									*											
Implementation									*											
Test Report																				
User Documentation																				
Week number	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Table 35: Actual project timeline

(* Easter)
Deadline: June 10th, 2008

The first notable difference is the extension of the COTS analysis. The final decision to use lwIP is marked with **x**; the date was March 14th. The reason for this extended period can be found by investigating the COTS process outlined in chapter 5.2. It is necessary to have a good understanding of the system requirements, and do a thorough search for potential components. It took time to reach the point where I knew enough about the desired functionality, and had learned how to search for components properly.

The requirement specification has been stable since the indicated time **rs1,** only minor changes have been applied since then. The changes were related to adding references to the proper standards, using correct terminology and similar things. The desired functionality has been constant during the project period.

The timeline also shows that design, testing and implementation has been done in parallell. Instead of trying to implement the whole design at once I developed PBMS in increments.

11.3.1 COTS Process

The COTS process that was followed in this project is outlined in chapter 5.2.1. I have in effect carried out a single iteration, and this report is part of the model evaluation stage. The end result will be found when the stack has been ported and validated for use in the target system. The desired functionality seems to be in place, but the quality attributes must be examined closely.

I have learned about many of the involved technologies for the first time in this project. Therefore trying to find suitable components before much of the problem was properly understood did not go so well. The COTS analysis was not completed before March 14th, when the final decision to use lwIP was made. The original plan was to do the analysis during the first two weeks of the project.

The results in chapter 10 show that very much of the functionality in PBMS is reused from lwIP. It is hard to say how this project would have gone if I was to build a stack from scratch, but my own estimate is that it I would not have come this far. By studying the design and implementation of lwIP I have become a better programmer than I was six months ago. I doubt that I back then would have been able to implement something similar to PBMS from scratch. Now I at least know where to start, what kind of challenges that must be solved, and where I can find comparable solutions.

The COTS process described in chapter 5.2.1 is essentially a search; the process ends when one suitable solution has been found. There may be several available components out there that could have done the job better than lwIP. Given the projects constraints the chosen component is to the best of my knowledge the most suitable of the found and investigated solutions.

11.3.2 Evaluation of the project plan

According to the initial plan, nearly all deliveries were finished too late. I made the project plan at the beginning of the semester, and did not know at the time how most of the individual tasks were to be solved. As long as the project is completed on schedule I would say the project plan has served its purpose.

Many software development methods take into account that the requirements may be altered frequently. This is one of the motivators for such methodologies as extreme programming [59]. This project has had stable requirements. I as a developer have had to learn quite a bit to understand the problem domain, but that is to be expected.

When requirements are stable the next step is to ensure that a design is implementable. The implementation was in essence a modification of existing code, so the test of implementability required careful studies of the existing source. Although it is beneficial to start prototyping early on in a project, I feel it could not have been done any sooner in this project.

11.4 Related work

11.4.1 A User-level Prioritization Service (UPS)

Ghias and Zeadally implemented a system with requirements somewhat similar to those of PBMS. The first of two papers describing the system is [60] “Enabling User Prioritization of Multimedia Applications”. As the title suggests, the application area is quite different from that of PBMS, but some of the applied techniques are the same. Their seconds article which describes the same system is [61], “Design and implementation of a User-level Prioritization Service”.

11.4.2 The UPS design problem

The fundamental problem they set out to solve is as follows: A user on a regular desktop operates several networked multimedia applications simultaneously. Different types of multimedia applications ideally require different levels of service quality. The user should be able to define the Quality of Service for each separate application. Their solution is the User-level Prioritization Service (UPS), which is implemented on Windows NT/2000/XP. UPS runs in user space, and is an additional scheduling layer between the sockets used by the multimedia applications and the kernel.

UPS supports Weighted Fair Queuing (WFQ), Class Based Queuing, and Priority Queuing [61], and the documented test cases were done with the WFQ algorithm. The ultimate design goal and associated results from UPS are not directly comparable with what has been done with PBMS.

11.4.3 Priority Queuing in UPS

The authors briefly describe some scheduling performance results found using the Priority Queuing algorithm [60]. “*Class Based Queuing and Priority Queuing also show the effectiveness of UPS in prioritizing the order of execution of networked applications*.”

Their first cited reference for the Priority Queuing algorithm is [62]. However, that paper is dedicated to a description of another algorithm; Bandwidth Guaranteed Priority Queuing (BGPQ). BGPQ is a combination of WFQ and Priority Queuing. According to the algorithm descriptions given in [60] it seems that UPS does not use BGPQ, even though they do refer to it.

12 Conclusion and future work

PBMS has been implemented as a simple set of modifications to lwIP. It shows that strict prioritization is indeed possible. Further tests are required to determine if PBMS can give a deterministic worst-case RTT in the target system. The protocol stack running on the supervisory unit is only one of several factors which ultimately decide the worst-case RTT. A much improved average behavior for priority traffic is nonetheless shown to be attainable.

12.1 Future Work

12.1.1 Academic

For the academic community a proof-of-concept Priority Based Message Stack is now available. It was desktop tested on Linux, but there is some complexity related to bypassing the built-in Linux TCP/IP stack. PBMS is still platform generic, just as lwIP. The other available platform ports should therefore be investigated if PBMS is to be used for other student projects.

Projects using standard lwIP are certainly also an option, if for instance the embedded systems are to function as normal Internet hosts, or protocols such as TCP are required.

The current maintenance status of the platform port in question should be checked before any project is started. The Linux port was supposedly up to date, but there were architectural inconsistencies in some central parts of it. Such things can certainly be managed. Updating a port to be compatible with the latest distribution of lwIP is both a good way to become familiar with lwIP, and a nice contribution to the open source community.

There is a win32 port, though this may have the same complications as the Linux port, since there already is a TCP/IP stack on most Windows systems. For networked embedded systems the following ports may be interesting:

- The 6502 CPU.
- The C16x/ST10 uC. (supports lwIP Raw API only.)
- The RTXC operating system.
- The Xilinx Virtex-II PRO device with embedded PowerPC 405 Processor.
- Motorola Coldfire 5272 CPU running under Nucleus OS.
- TI TMS320C6000 DSP running under uC/OS-II.

12.1.2 For Nera

I will continue this project on Nera's target system. The OS-specific port is to be developed, PBMS will be integrated with the rest of the system, and large scale performance tests are to be carried out. If the tests show that the system is indeed successful it will potentially be deployed as a part of Nera's product. This will depend on the result of an evaluation of all functional and quality attributes of PBMS.

13 References

1. Bass, L., P. Clements, and R. Kazman, *Software architecture in practice*. 2003, Boston, Mass.: Addison-Wesley. XXII, 528 s.
2. *IEEE standard glossary of software engineering terminology.* IEEE Std 610.12-1990, 1990.
3. Caro, D., *Automation network selection*. 2004, Research Triangle Park, N.C.: ISA-The Instrumentation, Systems, and Automation Society. XIII, 161 s.
4. *lwIP Savannah*. [cited 2008 31/5]; Available from: http://savannah.nongnu.org/projects/lwip/.
5. *lwIP ScribbleWiki*. [cited 2008 31/5]; Available from: http://lwip.scribblewiki.com/LwIP_Main_Page.
6. Dunkels, A. *lwIP - Links*. [cited 2008 2/6]; Available from: http://www.sics.se/~adam/lwip/links.html.
7. *The Open Group Base Specifications Issue 6*. 2004 [cited 2008 29/5]; Available from: http://www.unix.org/single_unix_specification/.
8. *Mailing list archive of lwIP-users*. [cited 2008 6/6]; Available from: http://lists.nongnu.org/archive/html/lwip-users/.
9. *IEEE recommended practice for software requirements specifications.* IEEE Std 830-1998, 1998.
10. Postel, J. *RFC768 User Datagram Protocol*. 1980 [cited 2008 13/5]; Status: STANDARD]. Available from: http://tools.ietf.org/html/rfc768.
11. Croft, W.J. and J. Gilmore. *RFC951 Bootstrap Protocol*. 1985 [cited 2008 13/5]; Status: DRAFT STANDARD]. Available from: http://tools.ietf.org/html/rfc951.
12. Wimer, W. *RFC1542 Clarifications and Extensions for the Bootstrap Protocol*. 1993 [cited 2008 13/5]; Status: DRAFT STANDARD]. Available from: http://tools.ietf.org/html/rfc1542.
13. Plummer, D. *RFC826 Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware*. 1982 [cited 2008 13/5]; Status: STANDARD]. Available from: http://tools.ietf.org/html/rfc826.
14. Postel, J. *RFC791 Internet Protocol*. 1981 [cited 2008 13/5]; Status: STANDARD]. Available from: http://tools.ietf.org/html/rfc791.
15. Hornig, C. *RFC894 A Standard for the Transmission of IP Datagrams over Ethernet Networks*. 1984 [cited 2008 13/5]; Status: STANDARD]. Available from: http://tools.ietf.org/html/rfc894.
16. Royce, W.W., *Managing the development of large software systems: concepts and techniques*, in *Proceedings of the 9th international conference on Software Engineering*. 1987, IEEE Computer Society Press: Monterey, California, United States.
17. *Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model.* ISO/IEC standard 7498-1:Second edition 1994-11-15, 1994.
18. *LMSC, LAN/MAN Standards Committee (Project 802)*. [cited 2008 29/5]; Available from: http://www.ieee802.org/.
19. Jha, S. and M. Hassan, *Engineering Internet QoS*. 2002, Boston: Artech House. xx, 325 s.
20. Nichols, K., et al. *RFC2474 Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*. 1998 [cited 2008 13/5]; Status: PROPOSED STANDARD]. Available from: http://tools.ietf.org/html/rfc2474.

21. Almquist, P. *RFC1349 Type of Service in the Internet Protocol Suite*. 1992 [cited 2008 13/5]; Status: PROPOSED STANDARD, Obsoleted by RFC2474]. Available from: http://tools.ietf.org/html/rfc1349.
22. Chiang, M.-L. and Y.-C. Li, *LyraNET: A zero-copy TCP/IP protocol stack for embedded systems.* Real-Time Syst., 2006. **34**(1): p. 5-18.
23. Dunkels, A. *Full TCP/IP for 8-Bit Architectures*. in *The First International Conference on Mobile Systems, Applications, and Services (MOBISYS `03)*. 2003. San Francisco, California.
24. Dunkels, A., et al., *Protothreads: simplifying event-driven programming of memory-constrained embedded systems*, in *Proceedings of the 4th international conference on Embedded networked sensor systems*. 2006, ACM: Boulder, Colorado, USA.
25. Dunkels, A., *Programming Memory-Constrained Networked Embedded Systems.* 2007. p. 214.
26. *Fusion TCP/IP Stack*. [cited; Available from: http://www.unicoi.com/fusion_net/fusion_tcp_ip.htm.
27. Interniche. *NicheStack IPv4*. [cited 2008 1/5]; Available from: http://www.iniche.com/nichestack.php.
28. Schantz, R.E., et al., *Controlling quality-of-service in distributed real-time and embedded systems via adaptive middleware: Experiences with Auto-adaptive and Reconfigurable Systems.* Softw. Pract. Exper., 2006. **36**(11-12): p. 1189-1208.
29. *The ACE ORB (TAO)*. [cited 2008 1/5]; Available from: http://www.theaceorb.com/.
30. *ACE+TAO Footprint Metrics Results*. [cited 2008 1/5]; Available from: http://www.dre.vanderbilt.edu/stats/footprint.shtml.
31. *Green Hills Software - Integrity RTOS features*. [cited 2008 1/5]; Available from: http://www.ghs.com/products/rtos/integrity_rtos_features_tools.html.
32. *Mailing list archive of lwIP-devel*. [cited 2008 6/6]; Available from: http://lists.nongnu.org/archive/html/lwip-devel/.
33. Bowman, I.T., R.C. Holt, and N.V. Brewster, *Linux as a case study: its extracted software architecture*, in *Proceedings of the 21st international conference on Software engineering*. 1999, IEEE Computer Society Press: Los Angeles, California, United States.
34. *SWAG: Software Architecture Group* [cited 2008 31/5]; Available from: http://www.swag.uwaterloo.ca/swagkit/.
35. Braden, R. *RFC1122 Requirements for Internet Hosts -- Communication Layers*. 1989 [cited 2008 2/6]; Status: STANDARD]. Available from: http://tools.ietf.org/html/rfc1122.
36. Braden, R. *RFC1123 Requirements for Internet Hosts - Application and Support*. 1989 [cited 2008 2/6]; Status: STANDARD]. Available from: http://tools.ietf.org/html/rfc1123.
37. Parnas, D.L., *Designing software for ease of extension and contraction*, in *Proceedings of the 3rd international conference on Software engineering*. 1978, IEEE Press: Atlanta, Georgia, United States.
38. *Doxygen*. [cited 2008 5/6]; Available from: http://www.stack.nl/~dimitri/doxygen/.
39. *Systems and software engineering - Recommended practice for architectural description of software-intensive systems.* ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15, 2007: p. c1-24.
40. Minghai, X., et al. *Implementation techniques of IntServ/DiffServ integrated network*. in *Communication Technology Proceedings, 2003. ICCT 2003. International Conference on*. 2003.

41. Kos, A., B. Klepec, and S. Tomazic. *Real-time application performance in differentiated services network*. in *Electrical and Electronic Technology, 2001. TENCON. Proceedings of IEEE Region 10 International Conference on*. 2001.
42. Schmitt, J. and F. Zdarsky. *A case for simplicity in providing network quality of service: class-based strict priority queueing*. in *Networks, 2004. (ICON 2004). Proceedings. 12th IEEE International Conference on*. 2004.
43. Ferrari, T., G. Pau, and C. Raffaelli, *Priority Queueing Applied to Expedited Forwarding: A Measurement-Based Analysis*, in *Proceedings of the First COST 263 International Workshop on Quality of Future Internet Services*. 2000, Springer-Verlag.
44. *Check: A unit test framework for C*. [cited 2008 1/6]; Available from: http://check.sourceforge.net/.
45. Fowler, M., *UML distilled: a brief guide to the standard object modeling language*. 2004, Boston, Mass.: Addison-Wesley. XXX, 175 s.
46. Miles, R. and K. Hamilton, *Learning UML 2.0*. 2006: O'Reilly Media, Inc.
47. *IEEE recommended practice for software design descriptions.* IEEE Std 1016-1998, 1998.
48. *Wireshark*. [cited 2008 5/6]; Available from: http://www.wireshark.org/.
49. Rönngren, R. and R. Ayani, *A comparative study of parallel and sequential priority queue algorithms.* ACM Trans. Model. Comput. Simul., 1997. **7**(2): p. 157-209.
50. *BSD Sockets Interface Programmer's Guide*. [cited 2008 3/5]; Available from: http://docs.hp.com/en/B2355-90136/.
51. Stevens, W.R., B. Fenner, and A.M. Rudoff, *UNIX Network Programming Vol. 1: The sockets networking API*. 3rd ed. 2004. XXIII, 991 s.
52. Boehm, B.W., *Software engineering economics*. 1981, Englewood Cliffs, N.J.: Prentice-Hall. XXVII, 767 s.
53. Intel. *Intel 64 and IA-32 Architectures Software Developer's Manuals*. [cited 2008 7/6]; Available from: http://developer.intel.com/products/processor/manuals/index.htm.
54. *Intel® Software Network - Q&A: RDTSC to measure performance of small # of FP calculations*. [cited 2008 7/6]; Available from: http://softwarecommunity.intel.com/isn/Community/en-US/forums/thread/30226599.aspx.
55. Almes, G., S. Kalidindi, and M. Zekauskas. *RFC2681 A Round-trip Delay Metric for IPPM*. 1999 [cited 2008 22/5]; Status: PROPOSED STANDARD]. Available from: http://tools.ietf.org/html/rfc2681.
56. Almes, G., J. Mahdavi, and M. Mathis. *RFC2330 Framework for IP Performance Metrics*. 1998 [cited 2008 22/5]; Status: INFORMATIONAL]. Available from: http://tools.ietf.org/html/rfc2330.
57. Almes, G., S. Kalidindi, and M. Zekauskas. *RFC2679 A One-way Delay Metric for IPPM*. 1999 [cited 2008 22/5]; Status: PROPOSED STANDARD]. Available from: http://tools.ietf.org/html/rfc2679.
58. *Linux Kernel Documentation :: networking : tuntap.txt*. [cited 2008 22/5]; Available from: http://www.mjmwired.net/kernel/Documentation/networking/tuntap.txt.
59. Beck, K. and C. Andres, *Extreme programming explained: embrace change*. 2004, Boston: Addison-Wesley. XXII, 189 s.
60. Ghias, S. and S. Zeadally. *Design and implementation of a user-level prioritization service*. in *Computers and Communication, 2003. (ISCC 2003). Proceedings. Eighth IEEE International Symposium on*. 2003.

61. Ghias, S. and S. Zeadally. *Enabling user prioritization of multimedia applications*. in *Information Technology: Coding and Computing [Computers and Communications], 2003. Proceedings. ITCC 2003. International Conference on*. 2003.
62. Law, K.L.E. *The bandwidth guaranteed prioritized queuing and its implementations*. in *Global Telecommunications Conference, 1997. GLOBECOM '97., IEEE*. 1997.