A lightweight and high-performance TCP/IP stack for Topsy

Semester Thesis of David Schweikert

April 1998 – July 1998

Computer Engineering and Networks Laboratory, ETH Zürich
Supervisor: George Fankhauser
Professor: Bernhard Plattner

Abstract

Topsy is a portable micro-kernel operating system designed for teaching purposes at the ETH Zürich. Goal of this project was the design and implementation of a TCP/IP Stack for Topsy. The whole networking infrastructure in Topsy had to be built.

Simplicity and readability were considered very important and therefore a modular, user-space protocol stack was designed. Efficiency was however preserved, with the use of techniques such as zero-copy operation and fast buffers. Each protocol module was kept completely independent of each other, with the use of "attributes" and a common configuration library which contains the dependence between the modules. The integration of new protocol modules should be very simple.

Each module was implemented with a Topsy thread. Although it is elegant to reuse Topsy's threading facilities, it is also too much complicated and expensive for this scope (pre-emptive multitasking). A simple user-space cooperative multitasking facility could be implemented in the future to enhance the performance, which is already good.

This report presents the project, analyses it and also serves as documentation for the implementation.

Zusammenfassung

Topsy ist eine portables Microkernel Betriebssystem, das am ETH Zürich für den Unterricht entworfen wurde. Ziel dieser Projekt war eine TCP/IP Stack für Topsy zu entwickeln. Die ganze Netzwerk Protokolle Infrastruktur in Topsy musste implementiert werden.

Wichtige gewünschte Eigenschaften waren Einfachheit und Lesbarkeit. Darum war eine user-space modulär Architektur gewählt, ohne aber zu viel an performance zu verlieren. Gute performance war erreicht mit “zero-copy” und schnelle buffers. Um den Design elegant zu machen, alle Modulen sind komplett unabhängig voneinander. Diese Unabhängigkeit ist erzielt mit “Attributen” und eine externe kleine “configuration library” der alle Abhängigkeiten enthält, so dass die Integration von neue Protokoll Modulen sehr einfach ist.

Jede Modul ist eine Topsy thread, so dass die Implementation elegant und einfach ist. Topsy threads sind aber zu kompliziert und ineffizient für diese Applikation (pre-emptive multitasking). Eine einfache “leightweight user-space cooperative multitasking” könnte in Zukunft implementiert werden um den Protokoll Stack schneller zu machen, der aber schon ziemlich effizient und klein ist.

Diese Bericht präsentiert und analysiert den Projekt. Es ist auch die Dokumentation für die Implementation.

Inhaltsverzeichnis

Chapter 1

Introduction

1.1 Project Goal

Topsy is a portable micro-kernel operating system designed at the ETH Zürich. Goal of this project was the design and implementation of a TCP/IP Stack for the Topsy operating system. The whole networking infrastructure in Topsy was to be designed.

1.2 Design Principles

Because it was designed as an educational software, very important properties of Topsy are it's simplicity and clean implementation which should also be very important for the TCP/IP Stack. Further documentation on Topsy can be found in [2] or at `http://www.tik.ee.ethz.ch/˜topsy`.

The TCP/IP Stack for Topsy was therefore designed with the following priorities in mind:

1. Readability and simplicity
2. Flexibility
3. Efficiency

Since in Topsy there is nothing to support networking protocols and devices, the full networking implementation, independent of TCP/IP, was developed. This was the major part of the work, and not the TCP/IP stack itself. We will however refer to the whole work as “TCP/IP Stack”, since that protocol stack is the one which we are interested in.

To support the the readability and flexibility goal without having significant efficiency penalties, several techniques were used, as described in chapter 2.

1.3 Development environment

Topsy was developed initially to be run on a IDT MIPS R3052E based board. A simulator for this platform (the “MipsSimulator”) was also written in java to simplify the development process of Topsy. Topsy without networking protocols and the MipsSimulator were already implemented and did constitute the basis for this project to build upon.

Prefix	**Files**	**Description**
`net`	`NetInit.c NetMain.c`	Generic/public functions (netInit)
`netbuf`	`NetBuf.c`	Network Memory Buffers
`netattr`	`NetAttr.c`	Network Attributes
`netcfg`	`NetConfig.c`	Configuration and classification of Packets
`netmod`	`NetModules.c`	Network Modules Interface
`netdbg`	`NetDebug.c`	Debugging Facility
`netif`	`NetIface.c`	Network Interfaces configuration/initialisation
`nettap`	`Ethertap` (directory)	Ethertap module
`neteth`	`Ethernet` (directory)	Ethernet module
`netip`	`IP` (directory)	IPv4 module
`netarp`	`ARP` (directory)	ARP module
`netudp`	`UDP` (directory)	UDP module
`neticmp`	`ICMP` (directory)	ICMP module

Table 1.1: Code Overview

Since on these embedded devices there isn't any network card, a simulated network device was added to the MipsSimulator: the "Ethertap" device, which is described in detail in appendix C.

The MipsSimulator was run on a Linux machine so that an interoperability testing was possible. It was also easier to debug the interface output, because on Linux there are already very good tools for this purpose such as `tcpdump`.

To make the evaluation of the performance and the debugging easier, a symbolic tracer and a profiler were added to the MipsSimulator. See appendix C for further information.

1.4 Code Overview

The TCP/IP stack will be in future distributed as a "plug-in", therefore it was not fully integrated into the main Topsy source distribution. Apart from small modifications in the `User/Makefile` and the kernel drivers, all the networking implementation resides in the `Net` directory.

To improve readability, a very strict prefixes convention was used for each subsystem of the networking implementation. The table 1.1 is a summary of all the prefixes used and the corresponding sources.

1.5 Acknowledgements

I would like to thank the TIK department at the ETH Zürich for giving me the chance to do this project, especially George Fankhauser, my project advisor. Without his suggestions and help all of this wouldn't have been possible. Thank you.

Chapter 2

Architecture

The architecture of the networking support in Topsy is now presented. This chapter should serve as a rapid overview of the major design decisions, which are important to understand before going into the implementation details.

2.1 User Space

The User-space or Kernel-space placement of non-core parts of an operating system is one of such problems which might never have a definitive answer and which are the cause of many “holy” wars on Usenet.

Since the main goal of Topsy is readability and not speed, a User-space implementation, which is certainly better understandable and easier to implement, was written.

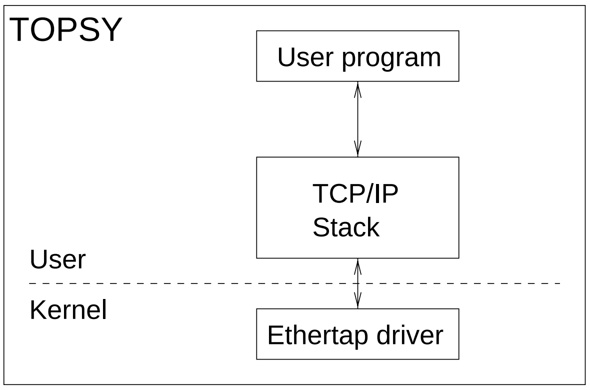

Figure 2.1: TCP Stack in User Space

2.2 Modularity

The fastest networking implementation is done with a ‘vertical’ (monolithic) approach, where the full processing of networking devices and protocols is done in one pass with efficient optimisations. However, such implementations are very complicated and inflexible. Implementing new protocols in such a framework is difficult.

A “modular” approach makes the implementation more flexible. Modules implement specific tasks of the protocol stack and should be the most independent to each other as possible. In such a framework, the implementation of new protocols is much easier.

Therefore a modular design was implemented, as described in chapter 5.

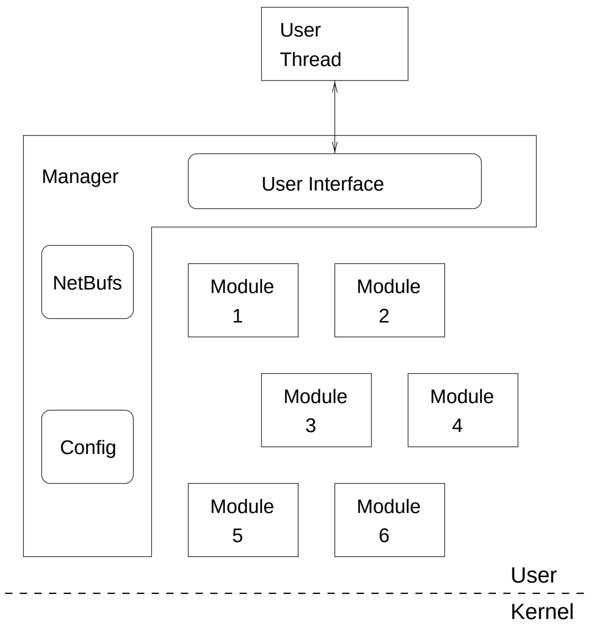

Figure 2.2: Modules architecture

2.3 Granularity

How small should a module be?

As we will see later, an overhead is added for each module to module interaction. That is, for efficiency reasons we don’t want that a packet has to be processed by too much modules. On the other side, we still want the advantages of modularity. A good compromise has to be made. Example of modules are: IP, TCP, ICMP, etc. A IP-reassembly module is for example too fine grained and a TCP/IP module is too large grained.

2.4 Configuration

To make the system as flexible as possible, each module should know as little as possible about the other installed modules. However, each module on a layer must know about the modules on the next layer to decide which one should receive the packet next. We call this decision a “configuration” decision.

The independence of each module to each other was resolved by the use of a “unclean” library, which makes the decisions of which module is the next to process a packet. This library is also responsible for the “cooperation” between the modules, for example it does the mapping between Ethernet addresses and IP addresses (with the help of the `ARP` module) between `IP` and `Ethernet`. We call this library “unclean” because it is highly dependent on all the modules, opposed to the "clean" independent protocol modules. The advantage of using such a library is that only this library has to be changed whenever a new protocol module is added.

The `netcfg` library is discussed in chapter 7.

2.5 NetBufs

One of the most differing characteristic between TCP/IP implementations is the choice of how the network data is stored in memory. The performance of the networking protocols is directly related to the memory management scheme.

An important design decision was that a `vmAlloc` requires about 2500 cycles to complete, which can become a major bottleneck on a network implementation because of the high number of allocation/deallocations which have to be made.

Another problem is the fragmentation of the memory, which, because of the high number of alloc/free made by a networking implementation, can become very high, if traditional algorithms are used.

To resolve this problem, only big chunks of memory (called pools) are allocated with `vmAlloc`. These pools are then divided in equal sizes buffers thus avoiding also internal fragmentation and making the implementation simple. A pool (which is 4096 bytes big) can be divided in 1 to 32 yielding buffers from 124 to 4080 bytes.

Common buffer manipulation such as adding a header can be made without the need to move data, in a way similar to BSD buffers, thus making the zero-copy goal possible.

NetBufs are explained in detail in chapter 3.

2.6 Attributes

Yet another problem with the independence of modules is that it is very frequent that a module must know information already processed by a module which did come before. For example the `TCP` can't be made really independent and must know the `IP` address of the sender, the receiver and so on. This is traditionally resolved with the use of complicated interfaces different for each module-to-module communication.

Even with the usage of complicated interfaces, a problem subsists: the communication of parameters between two modules which make use of a tunnel of modules between them. Consider this example: the "Type Of Service" field of the IP header must be controlled by the User interface module, but isn't known by the transport-layer modules such as TCP. How can the User interface module say the IP header which TOS it wants?

To resolve the above two problems a clever idea found in Scout , "a communication-oriented operating system targeted at network appliances", is used: the Attributes (see [3] or the Home-page at `http://www.cs.arizona.edu/scout`). We will call these "Attributes" Network Attributes or in short NetAttrs.

A NetAttr is a pair (key, value). In addition to the data-buffer (a NetBuf), a hash-table of NetAttrs (sort of database of attributes) is also passed from module to module. These attributes can be read and modified by each module in the vertical path. A module simply doesn't care about attributes it doesn't know about. For example the IP module sets the attributes for source address, destination address and so on. These attributes are then read by the TCP module.

NetAttrs are described extensively in chapter 6.

Chapter 3

Network Memory Buffers (`netbuf`)

3.1 Introduction

One of the most differing characteristic between TCP/IP implementations is the choice of how the network data is stored in memory. The performance of the networking protocols is directly related to the memory management scheme.

An important design decision was that a `vmAlloc` requires about 2500 cycles to complete, which can become a major bottleneck on a network implementation because of the high number of allocation/deallocations which have to be made.

Another problem is the fragmentation of the memory, which, because of the high number of alloc/free made by a networking implementation, can become very high, if traditional algorithms are used.

3.2 Memory Pools

To minimise the number of `vmAlloc` which have to be made, only "pools" of 4096 bytes (can be easily modified) are allocated. Because of the fixed size of the pools, the external fragmentation is kept at a minimum.

To avoid also internal fragmentation, each pool is divided in equal size NetBufs. The number of divisions is determined at the creation of the pool and is a power of 2.

To know which NetBuf is used and which is free, the bit-field `used` of 32 bits is kept in the pool-header. In this bit-field, if a NetBuf is free, the corresponding bit is 0, and if it is used, the corresponding bit is 1. The bit-field begins at the right (LSB) of the `used` long word. If for example the buf is divided in two, only the last two bits are used.

The biggest NetBuf is thus 4096 bytes minus the pool-header (4080) and the smallest possible results from a pool minus the pool-header divided in 32 aligned NetBufs (124 bytes).

Figure 3.1 shows how a pool is organised. In that figure, only the last 8 bits of the `used` bit-field are shown. Note the space at the end of the pool: for efficiency reasons, it was chosen to align each NetBuf to 4 bytes addresses. Therefore the size of each NetBuf is down-rounded to a multiple of 4, leaving some unused space at the end of the pool.

The pool-header also contains the `prev` and `next` pointers, which are used to implement double-linked lists. At the moment, only an array of these double-linked lists is maintained: the `freepools`.

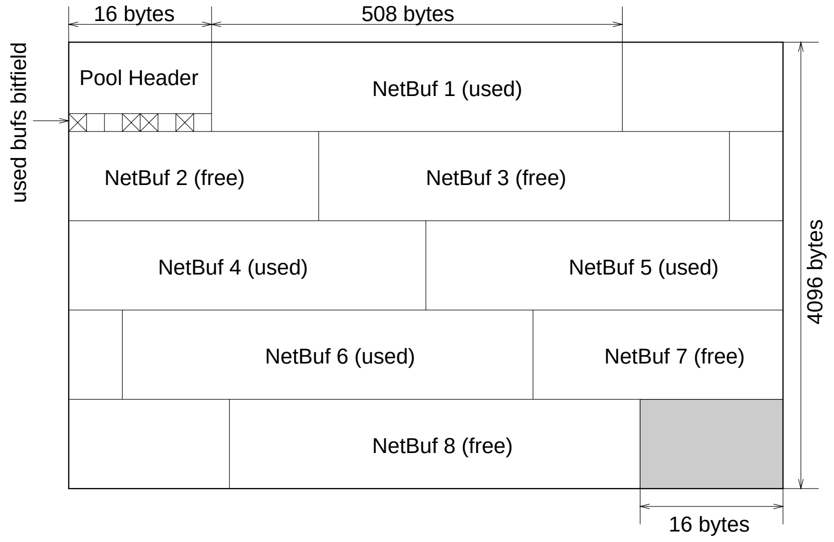

Figure 3.1: NetBufs Pool for 508-bytes NetBufs

The `freepools` array contains for each possible NetBuf size (for a `used` bit-field of 32 bits, the possible sizes are 6) a list of pools which have at least one free NetBuf which can be given away.

When the NetBuf-allocator is asked for a new NetBuf, the asked size is up-rounded and a pool with free NetBufs is searched in freepools. If none is found, a new pool is allocated and, unless the maximum size is asked, the new pool is added to the `freepools` array.

When a NetBuf is freed and the pool which contains it doesn't contain any other used NetBuf, the pool is also freed with a `vmFree`. To minimise the number of `vmAlloc` and `vmFree`, only when the `freepools` list of that size contains at least two entries, the pool is freed.

3.3 The NetBuf

The NetBufs are similar to the mbufs in BSD as described in [5], with the difference that mbufs are always 128 bytes long, whereas NetBufs can be from 124 to 4090 bytes long. In BSD, because of this limitation that all the mbufs are 128 bytes long, "clusters" where implemented, which are external 2048 (or 1024) bytes buffers. These "clusters" are however a sort of "trick", which is difficult to manage and not elegant. The dimension-flexibility of the NetBufs resolves this problem.

Figure 3.2 shows the structure of a NetBuf. `pool` is used to point back at the pool-header and `bufnr` is the position in the pool of this NetBuf. `size` is the dimension of the available payload space in this NetBuf (in bytes).

`next` is used to build linked lists of NetBufs, when for example a header has to be added, but there isn't enough space at the head of the NetBuf. All the networking functions should support the concatenation of NetBufs as if they were a unique contiguous NetBuf. The only assertion which can be made is that on the path from interface to user, all the headers are in the first NetBuf and on the path from user to interface no header of a layer is spread across two NetBufs. The NetBuf allocator makes also use of the `next` pointer in case the required buffer length is bigger that the biggest NetBuf, in which case a linked list is returned.

`start` and `end` are respectively offsets of the beginning and the end of the data in the NetBuf.

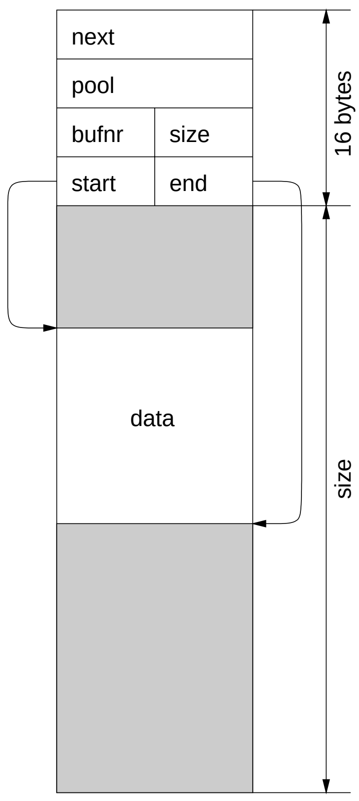

Figure 3.2: NetBuf structure

Please note that the shown structure and length of the header of the NetBufs or of the NetBuf pools could change in the future. Only the `netbuf` subsystem should access these headers directly. Other submodules should use the provided functions or macros.

3.4 NetBuf statistics

Some statistics are gathered in a structure named `netbufStats`. In this structure there are three arrays:

- `sizes`: for each size-index (from 0 to 5 for 32 bits `used` in pool-header) the corresponding real-size in bytes of the NetBuf (with header).
- `pools`: for each size-index the number of allocated pools.
- `usage`: for each size-index the number of allocated NetBufs.

Size-index goes from the most-divided pool to the non-divided pool.

A little Topsy-shell program (`NetStat.c`) is provided, which reads these variables and outputs on the console statistics like:

```
size    pools   netbufs memory  full
124     1       1       4k      3%
252     1       2       4k      12%
```

```
508     1       1       4k      12%
1020    1       1       4k      25%
2040    0       0       0k      0%
4080    0       0       0k      0%
```

3.5 The `netbuf` library

The speed advantage of the NetBufs is that no thread-switch is made for every alloc/free pair. This is possible because the NetBufs are managed cooperatively by each thread with a library of functions called the `netbuf` subsystem of library and which is written in `Net/NetBuf.c`.

There is a problem with this approach though: Topsy automatically throws away the regions which were allocated (`vmAlloc`) by a thread, when the thread is killed.

Consider this situation: thread A makes a alloc, which results in a `vmAlloc` of a new pool. Thread B makes also a alloc, and is given a buffer in the pool allocated by A. Now A for some reason is killed and with it the pool is freed. B still uses the buffer in the now-freed pool!

To resolve this problem, the `vmAlloc` and `vmFree` are always made by the `netmain` thread (described in section 8.2). When a networking thread wants to allocate a new pool, it sends a message (a `VMALLOC` message) asking the `netmain` thread to allocate it. Because of the design of NetBufs, the creation of new pools shouldn't happen frequently and therefore the added thread-switch shouldn't slow the system by much.

3.6 Macros description

Macro	**Description**
`NETBUF_MAXBUFS(s)`	Number of NetBufs in a pool for size-index `s`
`NETBUF_HEADERSIZE`	Size of NetBuf-header (at the moment 16 bytes).
`NETBUF_DATA(netbuf)`	Returns pointer to the data-region (`start` to `end`)
`NETBUF_SIZE(netbuf)`	Wrapper for field `size`
`NETBUF_START(netbuf)`	Wrapper for field `start`
`NETBUF_END(netbuf)`	Wrapper for field `end`
`NETBUF_LEN(netbuf)`	Data-region length (`start`−`end`)
`NETBUF_HEADSPACE(netbuf)`	Space available before `start`
`NETBUF_TAILSPACE(netbuf)`	Space available after `end`

3.7 Functions description

netbufInit

```
void netbufInit();
```

Initialises the `netbuf` subsystem.

netbufAlloc

```
int netbufAlloc(NetBuf *bufPtr, unsigned int size);
```

Allocates a new NetBuf with at least dimension `size`. If the required dimension is bigger than the biggest NetBuf possible, a linked list of NetBufs is allocated. If the allocation succeeds, the function returns the total size of the allocated NetBufs and `bufPtr` points to the new NetBuf (to the first in case of a linked-list). On fail, 0 is returned.

netbufFree

```
void netbufFree(NetBuf buf);
```

Frees the NetBuf `buf`. If `buf` is a linked list, frees all the NetBufs in the list.

netbufAddHead

```
int netbufAddHead(NetBuf *bufPtr, unsigned int len);
```

Function used to reserve space for a header to be added. If the NetBuf pointed by `bufPtr` has at least `len` bytes available before `start`, `start` is moved back by `len`. If there isn't enough space, a new (more if necessary) NetBuf is allocated and `*bufPtr` adjusted accordingly.

netbufLen

```
unsigned int netbufLen(NetBuf buf);
```

This function traverses the NetBuf list with head `buf` and sums up the total length of the data.

netbufTrim

```
void netbufTrim(NetBuf buf, unsigned int len);
```

If `buf`-data is longer than `len`, move `end` and, if necessary, remove NetBufs, so that the `buf`-data length is `len`.

netbufPosition

```
int netbufPosition(NetBuf *buf, unsigned int *position);
```

`position` refers to the absolute byte position in a virtual big buffer which is in reality a linked list of netbufs. After the function-call, buf points to the netbuf which contains that byte, and position is adjusted to the position in that netbuf. Returns 1 on success.

netbufCopy

```
int netbufCopy(NetBuf *dest, NetBuf src, int pos, int len);
```

Makes a copy of `src`-data starting from position `pos` and ending at `len`. It does traverse the linked list as if it were a single NetBuf. Returns 1 on success and 0 on failure. This function is used for the IP-fragmenting algorithm.

netbufClone

```
int netbufClone(NetBuf *dest, NetBuf src);
```

Makes an exact copy of the NetBuf `src`, including eventual NetBufs in a linked list. Returns 1 on success and 0 on failure.

Chapter 4

Network Device Drivers

4.1 Introduction

User-space protocol stacks are very nice, but they are traditionally also slow and therefore kernel-space implementation are more common today. It is however possible as described in [1] to achieve nearly same performance with user-space protocol stacks. The problem is that one more copy of data from kernel-space to user-space has to be made and more task switches also.

This implementation uses a mechanism which permits the direct copy of network interface data into a user-space buffer, thus removing the biggest disadvantage of user-space implementations.

The kernel-space drivers for the interfaces are kept intentionally the smallest possible. It is also possible that in the future, when a full paging memory management is implemented in Topsy, that all the driver will reside in user-space, because it will be then possible to access the interface memory from user-space.

4.2 Input processing

As shown in figure 4.1, an interface driver is composed of three parts: the interrupt service routine (ISR) in kernel-space, the interface-driver in kernel-space and the interface-module in user-space.

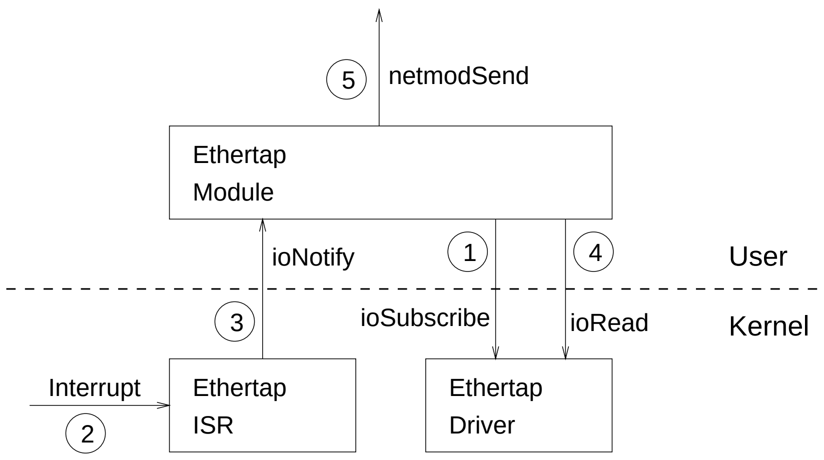

Figure 4.1: Ethertap input processing

It is very important that the ISR must complete the most rapidly possible, because since Topsy hasn't priority levels for interrupts, each ISR is fully masked, and thus the responsiveness of the whole system is compromised if they are not very fast.

The most simple protocol for the exchange of data would be (like the UART driver for example):

- User-space module makes a blocking `ioRead`
- When new data comes, the data is copied by the Interrupt Service Routine and the user-space module is unblocked.

The problem with this scheme is that the total length of the incoming packet isn't known when the user-space module makes the blocking `ioRead` before it's arrival (i.e. when the interrupt is triggered), and since we want to directly copy the data into user-space buffers, the following protocol is used (numbers as in figure 4.1:

1. The user-space module says to the kernel-space driver, that he his responsible for packets outgoing from this interface with `ioSubscribe`. Only the first module after the kernel-space driver initialisation is accepted.
2. When a packet arrives, an interrupt is triggered.
3. The Interrupt Service Routine sends a message to the registered user-space module saying that a new packet has arrived and how long it is (`ioNotify`).
4. The user-space modules allocates a new NetBuf, which can contain as much data as said in the `ioNotify` message and makes a non-blocking `ioRead` on the device.
5. The user-space module sends the data to higher-layer modules.

Output processing is much more simpler: a `ioWrite` is made whenever a new packet has to be sent.

4.3 Ethertap driver

The only driver which is implemented is the Ethertap driver, designed to work with the device described in section C.2.

The user-space module is described with the other modules in section 8.3.

4.3.1 Ethertap kernel-space driver (`IO/Drivers/Ethertap.c`)

The `ThreadId` of the user-space module (`subscribed`) is stored in the `extension` part of the `IODevice` structure which is initialised in `IO/IOMain.c` and which is then passed by Topsy to each kernel-space driver functions.

This file includes all the function which are required for the `IODeviceDesc` structure:

ethertap_interruptHandler

```
void ethertap_interruptHandler(IODevice this);
```

This function is the the Interrupt Service Routine (or “Handler”). It reads the packet length and sends a message to the user-space module saying that a message of that length is arrived and should be read from the device’s memory.

ethertap_init

```
Error ethertap_init(IODevice this);
```

Function called to initialise the kernel-space driver.

ethertap_read

```
Error ethertap_read(IODevice this, ThreadId threadId,
                    char* buffer, long int* size);
```

When a `ioRead` is made, this function is called, which copies `*size` bytes from the Ethertap device to `buffer`. Note that, although `buffer` will be probably contained in a NetBuf, the kernel-space doesn’t know anything about NetBufs and their structure, it just copies the data from `buffer` to `buffer+*size`.

ethertap_write

```
Error ethertap_write(IODevice this, ThreadId threadId,
                     char* buffer, long int* size);
```

When a `ioWrite` is made, this function is called, which copies `*size` bytes from `buffer` to the Ethertap device.

There is a commodity function called `netioWrite` in `NetIO.c` which follows the NetBuf and sends each chunk of data to the device.

ethertap_handleMsg

```
void ethertap_handleMsg(IODevice this, Message *msg);
```

This function is used to handle special messages: `IO_SUBSCRIBE` and `IO_GETADDR`.

`IO_SUBSCRIBE` writes the `ThreadId` of the sender thread in a device-specific variable, which is then used to send back `IO_NOTIFY` messages by the ISR.

`IO_GETADDR` is used to get the hardware address of the ethertap device (which is hard-coded in `Ethertap.java`).

Chapter 5

Modules

5.1 Overview

The modular architecture of the TCP/IP Stack was already described in section 2.2. Figure 5.1 gives a more detailed graphical overview.

5.2 Module Interface (`netmod`)

The definition of the interface of a module is contained in `NetModules.c` (and the corresponding header file).

A module is in Topsy a thread. This make the whole system much more simple and elegant because the message passing implementation is already implemented. It also makes a possible future dynamic addition of protocol modules at run-time possible. As we will see later, this has some costs though, which will be analysed in section 9.2.

`ThreadId`s of the module-threads are defined when the modules are created at run-time. Therefore another referencing scheme was implemented, so that each module has a unique fixed number. The mapping between this "`NetModuleId` and the `ThreadId` is made with a array which is filled at the creation of each module (see `netmodAdd`).

The `NetModuleId` for each module are defined in `NetModules.h` for the sake of simplicity (they have the form `NETMODULE_XXX`), but it is permitted to define new ones outside of it, provided that they are all equal or higher than `NETMODULE_LAST`.

A module can normally receive two type of messages: `NETMOD_SENDUP`, when a packet comes from a lower-layer module, and `NETMOD_SENDDOWN`, when a packet comes from a higher-layer module. Both of these messages share the same message structure, which is used for all the communications between modules and between a user module and a module:

```
typedef struct NetModMsg_t {
        ThreadId from;
        NetModMessageId id;
        NetBuf buf;
        NetAttrs attrs;
        long value;
} NetModMsg;
```

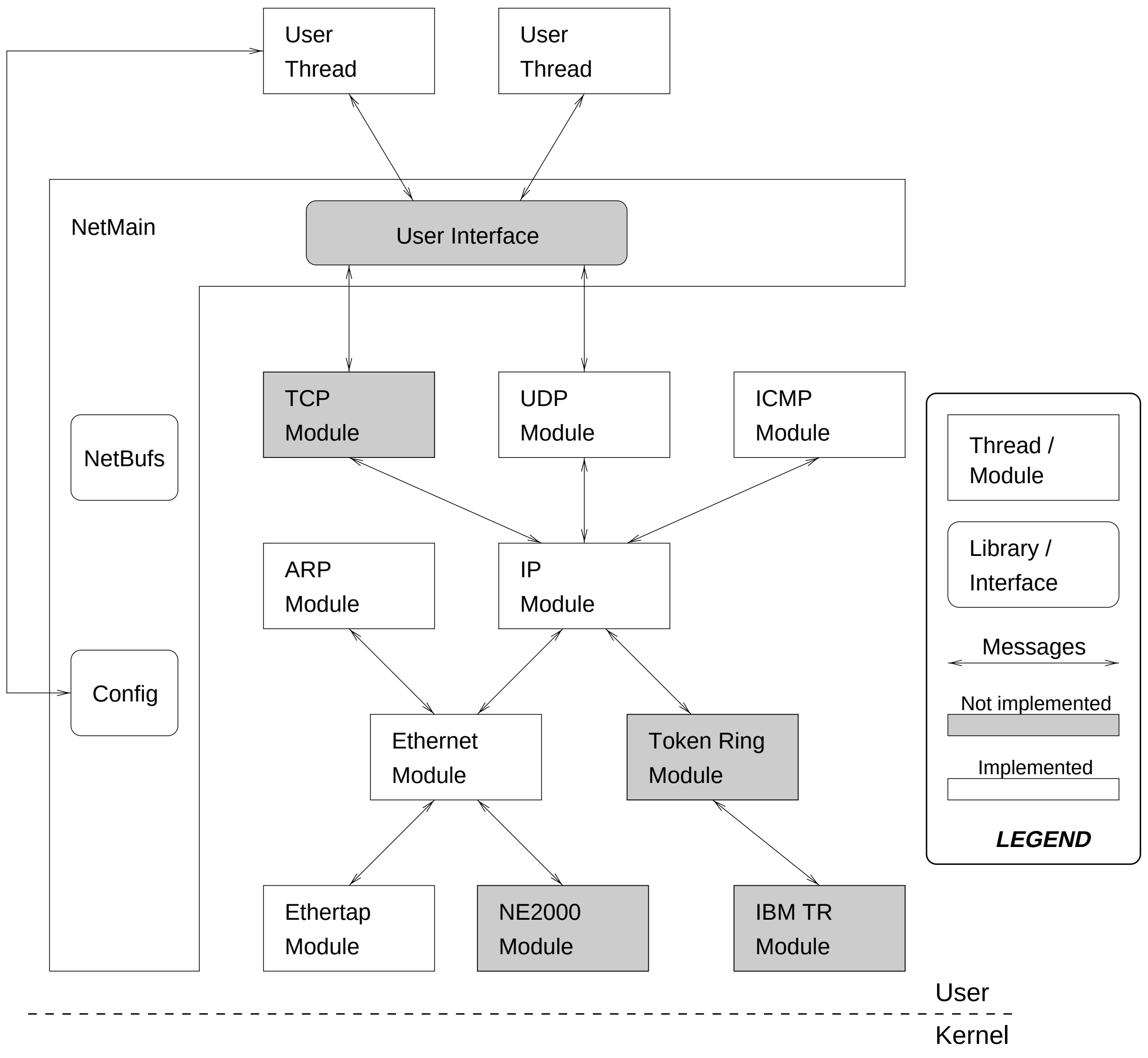

Figure 5.1: Modules overview

`from` and `id` are taken from the `Message` structure. Note that the dimension of a Message in Topsy at the moment is 4*5 bytes and `NetModSendMsg` was also made that long for compatibility reasons. `buf` is the NetBuf containing the packet-data and `attrs` is the table of attributes, which go along with the packet. `value` is used whenever only a long value has to be passed (it isn't used for `NETMOD_SENDUP` and `NETMOD_SENDDOWN`).

There are at the moment three more message types: `NETMOD_LISTEN` is used by a user thread to tell a module (at the moment only UDP) that it wants to receive a certain type of incoming packets (for the UDP the type is the destination port number, specified in the `value` message field, see section 8.8). `NETMOD_LISTENREPLY` is used by a module to acknowledge a `NETMOD_LISTEN` and `NETMOD_CLOSE` is used to close the "listening".

It is recommended to use the provided functions and not the structure directly. For each message type, two versions of the functions are provided: one which sends a message to a module identified by a `NetModuleId` (for example `netmodSendUp`) and one which sends the message directly to a thread identified by a `ThreadId` (for example `netmodSendUpThread`). The former is normally used to send a message to a module and the latter is used to send a message to a non-module thread (such as a user thread, which uses the networking modules).

5.3 Functions description

The `ThreadId` version of the functions aren't documented, because they are exactly the same as the `NetModuleId` versions, but are suffixed by `Thread` (ex. `netmodSendUpThread`) and their first parameter is a `ThreadId`.

netmodInit

```
void netmodInit();
```

Initialises the array which contains the mapping from `ThreadId` to `NetModuleId`.

netmodAdd

```
void netmodAdd(NetModuleId m, ThreadId t);
```

When a module is started, it has to call this function to register itself, so that the mapping between NetModuleId `m` and ThreadId `t` can be made.

netmodMsgRecv

```
int netmodMsgRecv(NetModMsg *msg);
```

This function is a simplified version of Topsy's tmMsgRecv. It returns 0 on failure. It is recommended to always use this function, because, if in the future a transition from the threaded structure of the modules to another could be made transparently.

netmodSendUp

```
void netmodSendUp(NetModuleId to,
                  NetBuf buf, NetAttrs attrs);
```

Sends to the module with NetModuleId `to` the packet contained in the NetBuf `buf` with attributes `attrs`. This function is called when a module has interpreted a packet (analysed and stripped the protocol header), which is now ready to be sent up to higher layer modules. Note that the `value` field is set always set to 0 for this message type.

netmodSendDown

```
void netmodSendDown(NetModuleId to, NetBuf buf,
                    NetAttrs attrs);
```

Sends to the module with NetModuleId `to` the packet contained in the NetBuf `buf` with attributes `attrs`. This function is called when a module has processed a packet (added the protocol header), which is now ready to be sent down to lower layer modules.

netmodListen

```
void netmodListen(NetModuleId to, NetAttrs attrs, long value);
```

Sends to the module with NetModuleId `to` a request to receive each packet which full fills certain criteria as specified in `attrs` and `value`.

netmodListenReply

```
void netmodListenReply(NetModuleId to, NetAttrs attrs,
                       long value);
```

Sends a response to the “Listen” request to the module with NetModuleId `to`. This function is currently not used, but is provided for consistency reasons (the reply is normally sent not to a module, but to a user thread).

netmodClose

```
void netmodClose(NetModuleId to, NetAttrs attrs, long value);
```

Sends to the module with NetModuleId `to` a request not to receive anymore packets which full fill certain criteria as specified in `attrs` and `value`.

Chapter 6

Network Attributes (`netattr`)

6.1 Introduction

As already said in section 2.6, a Network Attribute is a (key, value) pair, both `unsigned short`. An efficient way to store these attributes in a "database" was needed. This database (called NetAttrs) could however not take much memory, because for every packet, one of such databases is needed, and thus the memory footprint of the networking protocols would become too big.

For efficiency reasons a hash-table was the chosen implementation. The hash-table resides in a 252 bytes NetBuf. There can be maximally 59 attributes in such a hash-table as shown in figure 6.1. 252 bytes NetBufs were chosen because they are a good compromise to have an efficient but small hash-table. Note however that a different NetBuf-size could be chosen and nothing would have to be changed except `NetAttr.c` and the corresponding header file.

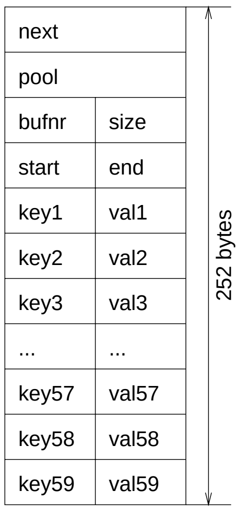

Figure 6.1: NetAttrs hash-table

Attributes are not used only for the message passing between modules, but are also used for the configuration of the interfaces and of the whole system, as described in chapter 7.

6.2 Functions description

netattrNew

```
NetAttrs netattrNew();
```

Allocates a new NetAttrs hash-table.

netattrFree

```
void netattrFree(NetAttrs a);
```

Frees the `a` NetAttrs hash-table. This is implemented with a macro.

netattrHash

```
int netattrHash(unsigned short key);
```

This is the hashing function. Shouldn't be used outside of `NetAttr.c`.

netattrFind

```
int netattrFind(NetAttrs a, unsigned short key);
```

Finds `key` in the hash-table `a`. Shouldn't be used outside of `NetAttr.c`.

netattrSet

```
int netattrSet(NetAttrs a, unsigned short key,
               unsigned short data);
```

Sets the attribute (`key`,`data`) in the hash-table `a`. Returns 1 on success, 0 on failure.

netattrUnset

```
void netattrUnset(NetAttrs a, unsigned short key);
```

Unset the key `key` in the hash-table `a`.

netattrGet

```
int netattrGet(NetAttrs a, unsigned short key,
               unsigned short *data);
```

Searches `key` in the hash-table `a` and, if it finds it, puts it's value intro `*data`. Returns 1 if `key` was found, 0 otherwise.

netattrDebug

```
void netattrDebug(unsigned long family, char *title,
                  NetAttrs a);
```

If family is enabled in `NetDebug.h`, prints a list of all the attributes present in the hash-table `a`. It prints also a short string (`title`) at the beginning.

6.3 List of attributes

The following table summarises all the attributes which are defined in `NetAttr.h`. Note however that new ones outside of `NetAttr.h` can be defined, provided that they are equal or superior to `NETATTR_LAST`.

Network Attribute	**Description**

`NETATTR_WANTED_LINK`
`NETATTR_WANTED_NETWORK`
`NETATTR_WANTED_TRANSPORT`
	These attributes are currently not used, but should in the future make it possible for a user-process (or a module) to specify with what modules the packet should be processed for each layer.

`NETATTR_IF_FROM` `NETATTR_IF_TO`	Interface attributes: `FROM` and `TO` are respectively the interface number (see section 7.2) from which comes the packet and to which it should go.
`NETATTR_IF_MTU`	Interface attribute: Maximum Transfer Unit for this interface (used only for the configuration of an interface).

`NETATTR_ETH_FROM_0`
`NETATTR_ETH_FROM_1`
`NETATTR_ETH_FROM_2`
`NETATTR_ETH_TO_0`
`NETATTR_ETH_TO_1`
`NETATTR_ETH_TO_2`
	Ethernet attributes: `FROM` and `TO` are the hardware addresses of respectively the sending host and the receiving host. They are specified with three attributes (0 is LSB) because the hardware address of a Ethernet device is 6 bytes long and an attribute is 2 bytes long. `FROM` is also used to specify the address of an interface.

`NETATTR_ETH_TYPE`	Ethernet attribute: network protocol-id in Ethernet header.

`NETATTR_IP_FROM_0`
`NETATTR_IP_FROM_1`
`NETATTR_IP_FROM_2`
`NETATTR_IP_FROM_3`
`NETATTR_IP_TO_0`
`NETATTR_IP_TO_1`
`NETATTR_IP_TO_2`
`NETATTR_IP_TO_3`
	Internet Protocol attributes: `FROM` and `TO` are the IP addresses of respectively the sending host and the receiving host. They are specified with four (could be made with two) numbers for commodity reasons. `FROM` is also used to specify the address of an interface.

`NETATTR_IP_TOS`	IP attribute: "Type Of Service" in IP header.
`NETATTR_IP_TTL`	IP attribute: "Time To Live" in IP header (maximum number of "hops" before the packet is discarded).
`NETATTR_IP_DF`	IP attribute: "Don't Fragment" in IP header (fragmenting shouldn't be made for this packet).
`NETATTR_IP_ID`	IP attribute: "Fragment ID" in IP header (used to reassemble fragments).
`NETATTR_IP_PROTOCOL`	IP attribute: "Protocol-ID" in IP header of the transport protocol above IP.

Network Attribute	Description
NETATTR_ARP_HW_TYPE NETATTR_ARP_HW_LEN NETATTR_ARP_PR_TYPE NETATTR_ARP_PR_LEN NETATTR_ARP_PR_0 NETATTR_ARP_PR_1	Address Resolution Protocol: These attributes are used for the generation of a ARP REQUEST message. TYPE, LEN, TYPE, LEN are the fields of the ARP request header (see [4]). 0 and 1 specify the IP address to be resolved.
NETATTR_UDP_FROM NETATTR_UDP_TO NETATTR_UDP_NOCHECKSUM	User Datagram Protocol: FROM and TO are respectively the source and destination port numbers. When NOCHECKSUM is set, the checksum won't be calculated (which is in UDP facultative, but suggested).
NETATTR_LAST	First attribute which can be used externally.

Chapter 7

Configuration

7.1 Introduction

The term “configuration” is here used for two different aspects of the TCP/IP Stack: the configuration of the host (for example the IP address, the interfaces and so on) and the configuration of the interaction between the modules (for example what modules are installed and what packet should go to each module).

We will will now see the first type of configuration: the configuration of the host.

7.2 Host and interfaces configuration

The same problems for the modularity of the implementation and the independence of the various modules apply to the configuration of the host: the configuration is highly dependent on the modules which are installed. I only have to configure the IP address of a interface if the IP module is installed. . .

In other words, each module wants to be able to retrieve information from a common configuration database. That is exactly the same function of Network Attributes, and thus were used also for this function, although were not initially thought to do so.

The host will have a NetAttrs hash-table for the global configuration and one for each interface with the per-interface configurations. At the moment the global configuration hash-table isn’t created, because there hasn’t been the need up to this writing. Only the per-interface configuration was done. All the routines for the configuration of the interfaces is in the subsystem `netif` (file `Net/NetIface.c`).

The `netif` subsystem not only does the configuration of the interfaces, but is also responsible for the initialisation and management of them. A interface is defined in a structure (`Net/NetIface.h`):

```
typedef struct NetIfDesc_t {
        unsigned int       nr;
        NetIfMainFunction  main;
        char               *name;
        NetModuleId        moduleId;
        NetAttrs           config;
} NetIfDesc;
```

Each interface is identified by it's interface number, which is also reproduced in the structure (`nr`), in case a pointer to the NetIfDesc structure is given, without the number. `main` is the main function of the interface driver module (thread). `name` is the name which should be given to the module thread and `moduleId` is the `NetModuleId` of the driver. `config` contains the attributes hash-table with the configuration.

The configuration is done at the moment at compile time in `Net/NetInit.c`, which is responsible for the initialisation of the whole networking system.

7.2.1 Functions description (`netif`)

netifInit

```
void netifInit();
```

Initialise each defined interface and start the corresponding driver module. The interfaces are defined in an global array (see `Net/NetIface.c`).

netifSetAttr

```
void netifSetAttr(unsigned int iface, unsigned short attr,
                  unsigned short data);
```

Set attribute `attr` to value `data` for interface `iface`.

netifGetAttr

```
int netifGetAttr(unsigned int iface, unsigned short attr,
                 unsigned short *data);
```

Get attribute `attr` for interface `iface` and put the value in `*data`. Return 0 on failure.

7.3 Modules Configuration (`netcfg`)

As said earlier (section 2.4), the glue between all the modules is done by an external subsystem (or library) which is called `netcfg` and which is contained mainly in `Net/NetConfig.c`.

For the moment, this "glue" work between the modules can be reduced to two important jobs: decide which is the next module to process the packet and do the necessary mapping between two different domains (for example map from Ethernet address to IP address).

The decision of which is the next module to receive the packet is done based only on the attributes. The scheme is as follows: each module interprete and fill the protocol headers contained in packet data (in a NetBuf) and based on these interpretations fill the attributes. The configuration library shouldn't modify or try to interprete the packet data directly, but should instead work only on the attributes. This is very important, because clearly leaves the implementation detail of the protocol only to the modules.

For this decision to succeed, it is thus important that each module fills enough attributes. For example it is mandatory that the IP module fills the `NETATTR_IP_PROTOCOL`, without which the configuration subsystem couldn't know if a packet should be sent to the UDP module, to the TCP module or another IP protocol module.

The mapping between two domains should be done as follows: the domain specific attributes which are also needed in the other domain, are translated according to a mapping and added to the attributes.

For example when a IP packet should be sent to a host with an Ethernet device, the attributes `NETATTR_IP_TO_x` (x from 0 to 3) are read by the configuration subsystem and the attributes `NETATTR_ETH_TO_x` (x from 0 to 2) are added according to the ARP cache (see the ARP module in section 8.5).

The implementation of the ARP cache is written in the file `Net/NetConfigARP.c`.

7.3.1 Functions description (`netcfg`)

netcfgInit

```
void netcfgInit();
```

Initialisation of the `netcfg` subsystem. For the moment only initialises the ARP cache.

netcfgSendNextDown

```
int netcfgSendNextDown(NetModuleId from, NetBuf buf,
                       NetAttrs attrs);
```

Every module should call this function when it has finished processing a packet which should go further down in the protocol layers. It sends the packet to the next module and does the mapping of attributes between the two modules if needed.

netcfgSendNextUp

```
int netcfgSendNextUp(NetModuleId from, NetBuf buf,
                     NetAttrs attrs);
```

Like `netcfgSendNextDown` but for packets in the opposite direction in the protocol layers.

netcfgarpAddCacheETHIP

```
void netcfgarpAddCacheETHIP(unsigned char *hw,
                            unsigned char *ip);
```

This functions is called by the ARP module (see 8.5) to add an entry to the ARP cache for the Ethernet/IP mapping. This function is defined in `Net/NetConfigARP.c`.

7.3.2 Private functions description (`netcfg`)

These functions shouldn't be used outside of the `netcfg` subsystem.

netcfgNextUp

```
NetModuleId netcfgNextUp(NetModuleId from, NetAttrs attr);
```

Return the next `NetModuleId` of the module that should process the packet which comes from module `from`, with attributes `attr` and which goes up in the protocol layers.

netcfgNextDown

```
NetModuleId netcfgNextDown(NetModuleId from, NetAttrs attr);
```

Same as `netcfgNextUp` but for packets in the opposite direction in the protocol layers.

netcfgTransformUp

```
int netcfgTransformUp(NetModuleId from, NetModuleId to,
                      NetBuf buf, NetAttrs attrs);
```

Make the required mapping for a packet which goes from module `from` to module `to`. In other words translate `from` specific attributes to `to`'s understandable attributes and add them to already existing `attrs`. `buf` is also passed, because it is possible that this packet is put on a hold queue to wait for a resolve-process to complete (currently this does never happen). In such a case, 0 is returned. If the packet can further be sent up, 1 is returned.

netcfgTransformDown

```
int netcfgTransformDown(NetModuleId from, NetModuleId to,
                        NetBuf buf, NetAttrs attrs);
```

Same as `netcfgNextUp` but for packets in the opposite direction in the protocol layers.

netcfgarpResolveETHIP

```
int netcfgarpResolveETHIP(NetBuf buf, NetAttrs attrs)
```

Do the mapping from a IP address to an Ethernet address for this packet. This function generates an ARP request in case one is needed (IP address not in cache). See the ARP module description in section 8.5 for further details. Returns 0 if the message was put in a hold queue and 1 if all is well. This function is defined in `Net/NetConfigARP.c`.

Note that since the hold queue isn't currently implemented, the first packet that generates an ARP request is simply thrown away. This is legal, because IP doesn't provide a guaranteed reliable service.

Chapter 8

Module Instances

8.1 Introduction

Each module shares the same structure, and thus, because of this similarity, each one won't be explained in detail.

To simplify the creation of new modules, a skeleton file was created, which is here reproduced, with some debugging functions call removed, to improve readability (`Net/Skel/Skel.c`):

```
#include <Topsy.h>
#include <Messages.h>
#include <Syscall.h>

#include <NetDebug.h>
#include <NetBuf.h>
#include <NetAttr.h>
#include <NetConfig.h>
#include <NetModules.h>

static void netsklMain(ThreadArg arg);
static void netsklUp(NetBuf buf, NetAttrs attrs);
static void netsklDown(NetBuf buf, NetAttrs attrs);

void netsklInit()
{
    ThreadId netsklId;

    /* Start main thread */
    tmStart(&netsklId,netsklMain,(ThreadArg)0,"netskl");
}

static void netsklMain(ThreadArg arg)
{
    ThreadId myThreadId, parentThreadId;
    NetModMsg msg;

    tmGetInfo(SELF, &myThreadId, &parentThreadId);

    /* Add to list of modules */
    netmodAdd(NETMODULE_SKEL, myThreadId);
    while(1) {
        if(!netmodMsgRecv(&msg)) return;

        switch (msg.id) {
        case NETMOD_SENDUP:
            netsklUp(msg.buf, msg.attrs);
            break;
        case NETMOD_SENDDOWN:
            netsklDown(msg.buf, msg.attrs);
            break;
        }
    }
}

static void netsklUp(NetBuf buf, NetAttrs attrs)
{
    /* Process packet and fill attrs */

    /* Send up packet */
    netcfgSendNextUp(NETMODULE_SKEL, buf, attrs);
}

static void netsklDown(NetBuf buf, NetAttrs attrs)
{
    /* Build header and fill attrs */

    /* Send down packet */
    netcfgSendNextDown(NETMODULE_SKEL, buf, attrs);
}
```

As you can see, the basic structure of a module is very simple: the init function starts the thread (and does some initialisation, if necessary). The main function is a never ending loop which processes the incoming messages. The "Up" function is called for upcoming packets and the "Down" function is called for down going packets. Both of these function process the packet and call `netcfgSendNextUp` or `netcfgSendNextDown`.

The init function is then called by the `netInit` function defined in `Net/NetInit.c`, where the initial configuration also happens.

Each module resides in a separate directory under `Net`. A short description of each module now follows, with some notes when there is something special about a modules.

8.2 The `netmain` thread

This isn't really a module, although in the future it could be also made so. The `netmain` thread is thought as the controlling thread. It receives pool creation requests (see chapter 3) and should in the future also be responsible for the user interface.

The code for the `netmain` thread resides in `Net/NetMain.c`

8.3 Ethertap Module (`nettap`)

This is a driver module for the Ethertap device, as explained in section C.2. It supports the standard `NETMOD_SENDDOWN` message, but not the `NETMOD_SENDUP` message because it doesn't make sense. It supports also the `IO_NOTIFY` message, as explained in section 4.2.

Required and Added attributes

	Up going packet	**Down going packet**
Required		`NETATTR_IF_TO`
Facultative		
Added	`NETATTR_IF_FROM`	

8.4 Ethernet Module (`neteth`)

Module which receives the packets from all the installed Ethernet device modules (such as Ethertap). Does interprete and fill the Ethernet header.

Required and Added attributes

	Up going packet	**Down going packet**

Required	`NETATTR_IF_FROM`	`NETATTR_IF_TO`
		`NETATTR_ETH_TO_0`
		`NETATTR_ETH_TO_1`
		`NETATTR_ETH_TO_2`
		`NETATTR_ETH_TYPE`

Facultative		

Added	`NETATTR_ETH_FROM_0`
	`NETATTR_ETH_FROM_1`
	`NETATTR_ETH_FROM_2`
	`NETATTR_ETH_TO_0`
	`NETATTR_ETH_TO_1`
	`NETATTR_ETH_TO_2`
	`NETATTR_ETH_TYPE`	

8.5 ARP module (`netarp`)

Address Resolution Protocol (see [4], pp. 53–63). When an ARP request is processed, this module directly generates the reply and calls a function in `netcfg` (`netcfgarpAddCacheETHIP`) to add the senders's ARP information. The `netcfg` itself can send messages to this module, with a null buffer, to generate ARP requests.

For the moment, only the ARP for IP on Ethernet is implemented.

Required and Added attributes

	Up going packet	Down going packet
Required	NETATTR_IF_FROM	NETATTR_IF_TO NETATTR_ARP_HW_TYPE NETATTR_ARP_HW_LEN NETATTR_ARP_PR_TYPE NETATTR_ARP_PR_LEN NETATTR_ARP_PR_0 NETATTR_ARP_PR_1
Facultative		
Added		NETATTR_ETH_TYPE NETATTR_ETH_TO_0 NETATTR_ETH_TO_1 NETATTR_ETH_TO_2

8.6 IP module (`netip`)

Internet Protocol (see [4], pp. 33–52). In the `IP` directory there is also a file (`NetIPchecksum.c`) which contains the IP checksum algorithm (taken from BSD and adapted to NetBufs) and which is also used by other IP protocol modules such as UDP.

The IP module isn't complete. It lacks for the moment forwarding and reassembly.

Required and Added attributes

	Up going packet	Down going packet
Required	NETATTR_IF_FROM	NETATTR_IP_TO_0 NETATTR_IP_TO_1 NETATTR_IP_TO_2 NETATTR_IP_TO_3 NETATTR_IP_PROTOCOL
Facultative		NETATTR_IP_ID NETATTR_IP_TOS NETATTR_IP_TTL NETATTR_IP_DF
Added	NETATTR_IP_FROM_0 NETATTR_IP_FROM_1 NETATTR_IP_FROM_2 NETATTR_IP_FROM_3 NETATTR_IP_TO_0 NETATTR_IP_TO_1 NETATTR_IP_TO_2 NETATTR_IP_TO_3 NETATTR_IP_PROTOCOL	NETATTR_IF_TO

8.7 ICMP module (`neticmp`)

Internet Control Message Protocol (see [4], pp. 69–83). For the moment only the ECHO REQUEST and ECHO REPLY (ping) messages are implemented.

Required and Added attributes

	Up going packet	**Down going packet**

Required	`NETATTR_IP_FROM_0`
	`NETATTR_IP_FROM_1`
	`NETATTR_IP_FROM_2`
	`NETATTR_IP_FROM_3`	

Facultative		

Added		`NETATTR_IP_TO_0`
		`NETATTR_IP_TO_1`
		`NETATTR_IP_TO_2`
		`NETATTR_IP_TO_3`
		`NETATTR_IP_PROTOCOL`

8.8 UDP module (`netudp`)

User Datagram Protocol (see [4], pp. 143–167). This module can receive `NETMOD_LISTEN` and `NETMOD_CLOSE` requests as described in section 5.2.

The checksum calculation is a problem because UDP needs the source IP address to calculate it and isn't known (if not specified) when the UDP wants to do it.

Required and Added attributes

	Up going packet	**Down going packet**

Required	`NETATTR_IP_FROM_0`
	`NETATTR_IP_FROM_1`
	`NETATTR_IP_FROM_2`
	`NETATTR_IP_FROM_3`
	`NETATTR_IP_TO_0`
	`NETATTR_IP_TO_1`
	`NETATTR_IP_TO_2`
	`NETATTR_IP_TO_3`
		`NETATTR_IP_FROM_0`
		`NETATTR_IP_FROM_1`
		`NETATTR_IP_FROM_2`
		`NETATTR_IP_FROM_3`
		`NETATTR_IP_TO_0`
		`NETATTR_IP_TO_1`
		`NETATTR_IP_TO_2`
		`NETATTR_IP_TO_3`
		`NETATTR_UDP_FROM`
		`NETATTR_UDP_TO`

Facultative		
Added	`NETATTR_UDP_FROM` `NETATTR_UDP_TO`	

Chapter 9

Results and Conclusions

9.1 Ping output and tcpdump

When run a PentiumPro 180Mhz machine with Linux, a ping to Topsy on the MipsSimulator looks as follows:

```
dave@schweikert:/home/dave > ping 192.168.2.2
PING 192.168.2.2 (192.168.2.2): 56 data bytes
64 bytes from 192.168.2.2: icmp_seq=0 ttl=64 time=485.8 ms
64 bytes from 192.168.2.2: icmp_seq=1 ttl=64 time=193.8 ms
64 bytes from 192.168.2.2: icmp_seq=2 ttl=64 time=200.4 ms
64 bytes from 192.168.2.2: icmp_seq=3 ttl=64 time=192.5 ms
64 bytes from 192.168.2.2: icmp_seq=4 ttl=64 time=193.4 ms

--- 192.168.2.2 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 192.5/253.1/485.8 ms
```

The output of `tcpdump` (a very nice network debugging tool) of the previous ping is:

```
schweikert:/home/dave # tcpdump -vv -i tap0
tcpdump: listening on tap0
11:13:39.598072 arp who-has topsy tell linux
11:13:39.838026 arp reply topsy is-at 1:2:3:4:5:6
11:13:39.838068 linux > topsy: icmp: echo request (ttl 64, id 5119)
11:13:40.083218 topsy > linux: icmp: echo reply (ttl 64, id 1)
11:13:40.595754 linux > topsy: icmp: echo request (ttl 64, id 5127)
11:13:40.789248 topsy > linux: icmp: echo reply (ttl 64, id 2)
11:13:41.595767 linux > topsy: icmp: echo request (ttl 64, id 5132)
11:13:41.795699 topsy > linux: icmp: echo reply (ttl 64, id 3)
11:13:42.595788 linux > topsy: icmp: echo request (ttl 64, id 5137)
11:13:42.787947 topsy > linux: icmp: echo reply (ttl 64, id 4)
11:13:43.595776 linux > topsy: icmp: echo request (ttl 64, id 5142)
11:13:43.788832 topsy > linux: icmp: echo reply (ttl 64, id 5)
```

9.2 Performance analysis

This section will try to evaluate the performance of the protocol stack. As a benchmarking test it was chosen an ICMP ECHO REQUEST coming from the Ethertap interface followed by a ICMP ECHO REPLY generated by Topsy and sent back. The test was made with a 56-bytes load (standard ping) and a 1000-bytes load.

To make the analysis easier, the MipsSimulator's Tracer was modified to write to a file each address of the executed instructions. This trace-file was then analysed by a program which did count for each function the number of executed instructions (profiling).

The full results of the profiling are in appendix D. From these results it is immediately evident that the change from 56 to 1000 bytes does change only two functions: longCopy, which is used to copy the data from the network interface to the NetBuf, and netipChecksum which is also used to compute the ICMP checksum (the IP checksum is made on the same amount of data: the IP header).

The table in the previous section can be so summarised:

	1000-bytes ping		**56-bytes ping**	
function	**ins**	**%**	**ins**	**%**
Read/Write from device	4465	17.63	1161	5.54
Interrupt Service Routine	78	0.31	78	0.38
Message passing and Scheduling	11524	45.55	11524	55.22
Network Modules	2775	10.95	1706	8.19
Network Attributes	2679	10.58	2679	12.83
NetBuf routines	801	3.17	734	3.52
Configuration (netcfg)	515	2.03	515	2.48
Miscellaneous	2468	10.66	2473	10.96

To process an echo request and generate the reply, more than half of the time (for 56 bytes ping) is spent in scheduling and messaging. Although some scheduling would have to be made also with a not multi threaded network protocol stack (kernel ISR switch to user protocol to kernel device read to user protocol to kernel device write), it would be certainly less.

Note that under Miscellaneous, there is the lock and unlock of spinlocks, the Idle thread and other minor functions.

In the following section, a possible solution for the problem of too much scheduling is proposed.

The networking code itself in the modules appears to be efficient. Network Attributes cost also something but it is acceptable, given the advantages that they provide. Some optimisations could be made on the attributes, with the caching of information which is retrieved for each packet from the originating interface (such as it's IP address). NetBuf allocation and deallocation is also very fast compared to normal vmAlloc figures (which include two scheduling decisions).

This is implementation is very small: the size of whole networking user-space (as described in this documentation) is 26 kbytes (text+data) and would be thus appropriate for an embedded device where the memory is much limited. The dynamic allocation of buffers (versus a pre-defined, always allocated buffers) makes also the smallest memory footprint possible at runtime.

9.3 Future possible developments

9.3.1 Completion of TCP/IP stack

Implementing a protocol stack is a huge work and by the implementation of this one, some compromises had to be made. Most noticeably the TCP protocol couldn't be implemented because of lack of time. Unfortunately TCP is also the most important IP protocol, so, if some serious usage of this protocol stack is to be made, the TCP module should be written.

Another possible extension is the IP forwarding which completely lacks in this implementation, but which isn't normally needed by hosts (not routers).

The modular approach of the whole networking implementation and the use of attributes should however make the writing of those parts easier than for example it was for BSD.

9.3.2 Other protocol stacks

TCP/IP is certainly nowadays the most important protocol stack, but protocols which build upon TCP/IP (such as SKIP for example) are becoming also very important. With this modular approach and the fact that it is all in user-space, high layer protocols such as FTP for example could also be implemented as modules.

The whole networking infrastructure is already implemented, and thus TCP shouldn't require much time to implement. A required service that TCP requires and that isn't implemented is the whole timers management (multiple threads should be able to request timer messages at different time intervals).

IPv6 is the successor of IP (version 4) and therfore is also very important and could also be implemented in this framework.

9.3.3 User interface

No user interface proper was implemented. For the moment, the user programs communicate directly to the networking modules, as if they were them self modules.

A global user interface which encapsulates all the modules and doesn't show to the user the internal organisation of the modules could be an advantage. It would mean easier network programming and also portability if the module change.

If a user interface is implemented, it would be a big advantage to make it, if not completely compatible, similar to the Unix Sockets, so that the porting of programs from other platforms would be easier.

9.3.4 Lightweight user-threads

As shown in section 9.2, the message passing between modules costs considerable time. To improve the performance without abandoning the modular concept and the ease of implementation, lightweight user-threads could be used. Such lightweight user-threads are cooperative tasks implemented completely in user-space, removing thus the great complexity of the preemptive multitasking (saving of all the registers and so on).

Appendix A

Project Description

A.1 Einleitung

Topsy ist ein portables Microkernel Betriebssystem, das am TIK für den Unterricht entworfen wurde. In der ersten Version wurde es für die Familie der 32-bit MIPS Prozessoren gebaut. Es zeichnet sich durch eine saubere Struktur, eine hohe Portabilität (Trennung des Systems in hardware-abhängige und -unabhängige Module) und eine gute Dokumentation [1] aus. Im weiteren wird das System auf die Familie der intel i386 Prozessoren portiert. Weitere Dokumentation über Topsy ist unter http://www.tik.ee.ethz.ch/ topsy verfügbar.

A.2 Aufgabenstellung

Obwohl bereits im Praktikum TI2 einfache Netzwerktreiber und Protokollmodule für Framing, Routing und Zuverlässigkeit entworfen werden, besteht keinerlei Einbindung des Systems in die Welt der Internet-Protokolle. In dieser Semesterarbeit geht es darum, nach der gleichen Philosophie nach der das Betriebssystem entworfen wurde (schlank, lesbar, schnell), einen Internet-Protokollstack zu entwerfen und zu implementieren.

A.2.1 Ziele

Das Hauptziel dieser Arbeit ist eine les- und brauchbare TCP/IP Implementation für das Betriebssystem Topsy. Dabei soll eine vollständige IPv4 Implementation mit ICMP angestrebt werden. Bezüglich trade-off Lesbarkeit/Performance soll das Design Aufschluss geben, wie weit eine TCP/IP Implementation klar strukturiert und modularisiert werden kann, ohne grosse Leistungseinbussen in Kauf nehmen zu müssen.

A.2.2 Vorgehen

Da es sich bei der vorliegenden Arbeit um eine sehr anspruchsvolle und aufwendige Arbeit handelt, soll die Arbeit, um möglichst effizient vorzugehen, in vier Phasen gegliedert werden.

Der erste Teil beschäftigt sich nur mit dem Design, dazu gehören u.a.:

- Lesen Sie sich in die Spezifikation von TCP/IP ein (RFC 791/793).

- Machen Sie sich mit dem Betriebssystem Topsy vertraut.
- Machen Sie sich mit Konzepten wie Zero-copy I/O, getrennte Daten- und Kontrollpfade, Modularisierung, Memory Management, Timer Management und Kernel- vs. Userspace Implementation vertraut.
- Analysieren Sie bestehende Ansätze (BSD, Karn, etc.) und zeigen Sie deren Vor- und Nachteile auf.

Die zweite Phase dient dazu, eine saubere Entwicklungs- und Testumgebung aufzubauen:

- Stellen Sie sich die notwendigen Tools auf einer Hostumgebung (z.B. Linux) zusammen. Dazu gehören gcc, gas, gld, objcopy (alle i386 oder cross-i386) und bash, gmake, java (plattformunabhängig).
- Es soll ein Pseudo-Packetdriver für den Simulator auf Basis von Raw-sockets angelegt werden.
- Ueberlegen Sie sich, welche "Testprogramme" sinnvoll sind: ping (IP/ICMP), telnetd oder remote shell (TCP), httpd (fake, versteht nur GET /), tcp_crash_you (random sender, attacker), andere?

Die Phase 3 legt den Grundstein der Implementation:

- Hier wird die Basisfunktionalität von IPv4 implementiert. Kontrollprotokolle wie ICMP gehören auch hier hinzu. Spezialitäten, wie IGMP, die nicht zwingend notwendig sind, können weggelassen werden.

In Phase 4 kann auf IP aufgebaut werden, hier soll die Transportschicht mit UDP und TCP implementiert werden.

Weitere Hinweise:

- Auf dem Simulator testen, optional native (bräuchte einen Network-Driver), auf alle Fälle mit einem andern Stack zusammen.
- Soll der Stack nicht nur auf dem Simulator gestestet werden, sondern auch mit Ethernet-Karten (optional, z.B. auf Topsy i386) muss das ARP (Address Resolution Protocol) für Ethernet implementiert werden.
- Name lookups werden üblicherweise mit dem DNS Protokoll durchgeführt. In erster Näherung soll jedoch nur eine lokale "resolve function" implementiert werden.
- Portabilität soll unbedingt erhalten bleiben (Bi-endian support) und Stabilität soll erreicht werden (geeignete Testmethode).

A.3 Bemerkungen

- Mit dem Betreuer sind wöchentliche Sitzungen zu vereinbaren. In diesen Sitzungen soll der Student mündlich über den Fortgang der Arbeit berichten und anstehende Probleme diskutieren.

- Am Ende der zweiten Woche ist ein Zeitplan für den Ablauf der Arbeit vorzulegen und mit dem Betreuer abzustimmen.
- Am Ende des zweiten Monats der Arbeit soll ein kurzer schriftlicher Zwischenbericht abgegeben werden, der über den Stand der Arbeit Auskunft gibt.
- Am Ende der zweiten Woche ist ein Zeitplan für den Ablauf der Arbeit sowie eine schriftliche Spezifikation der Arbeit vorzulegen und mit dem Betreuer abzustimmen.
- Bereits vorhandene Software kann übernommen und gegebenenfalls angepasst werden.
- Die Dokumentation ist mit dem Textverarbeitungsprogramm “FrameMaker” zu erstellen.

A.4 Ergebnisse der Arbeit

Neben einem mündlichen Vortrag von 20 Minuten Dauer im Rahmen des Fachseminars Kommunikationssysteme sind die folgenden schriftlichen Unterlagen abzugeben:

- Ein kurzer Bericht. Dieser enthält eine Darstellung der Problematik, eine Beschreibung der untersuchten Entwurfsalternativen, eine Begründung für die getroffenen Entwurfsentscheidungen, sowie eine Auflistung der gelösten und ungelösten Probleme. Eine kritische Wür-digung der gestellten Aufgabe und des vereinbarten Zeitplanes rundet den Bericht ab (in vierfacher Ausführung).
- Ein Handbuch zum fertigen System bestehend aus Systemübersicht, Implementationsbeschreibung, Beschreibung der Programm- und Datenstrukturen sowie Hinweise zur Portierung der Programme.
- Eine Sammlung aller zum System gehörenden Programme.
- Die vorhandenen Testunterlagen und -programme.
- Eine englischsprachige (Deutsch falls Bericht Englisch) Zusammenfassung von 1 bis 2 Seiten, die einem Aussenstehenden einen schnellen berblick über die Arbeit gestattet. Die Zusammenfassung ist wie folgt zu gliedern: (1) Introduction, (2) Aims & Goals, (3) Results, (4) Further Work.

A.5 Literatur

[1] G. Fankhauser, C. Conrad, E. Zitzler and B. Plattner., Topsy - A Teachable Operating System, TIK, 1997

[2] Topsy home page: http://www.tik.ee.ethz.ch/ topsy

[3] RFC 791: IP

[4] RFC 793: TCP

Appendix B

Debugging facilities (netdbg)

There was a need for a flexible debugging facility inside of the networking implementation, which could be selective on which debugging information to display.

Each subsystem has a corresponding `NETDEBUG_XXX` ‘family-id’. `NETDEBUG_MASK` is set (with OR) to each ’family-id’ of the subsystems for which the debugging information is wanted (see `NetDebug.h`).

For example:

```
#define NETDEBUG
#define NETDEBUG_MASK (NETDEBUG_ETHERNET | NETDEBUG_IP \
                      | NETDEBUG_ICMP)
```

will make the debug information of the Ethernet, IP and ICMP subsystems display on screen. To turn off all the debugging information, `NETDEBUG` can be undefined.

B.1 Functions description

netdbgInit

```
void netdbgInit();
```

Initialises the netdbg subsystem.

netdbgDisplay

```
void netdbgDisplay(unsigned long family, char *str);
```

Displays `str` if `family` is selected.

netdbgPrintf

```
void netdbgPrintf(unsigned long family, char *fmt, ...);
```

This is a function similar to the C-library function `printf`, but only displays on screen if `family` is selected.

To make the implementation more flexible, the netdbgPrintf uses the function `vsprintf` which can be found in `User/UserSupport.c`. Use this function and all the other functions which make use of `vsprintf` with caution, because `vsprintf` is far from complete and stable.

Appendix C

Development Environment

C.1 Linux Ethertap device

Since I had already Linux installed at home and since it is very flexible, I've chosen Linux as the development platform, on which I would then run the MipsSimulator.

Linux can be configured to simulate a network device which is called "Ethertap". It is so described in it's documentation:

```
Ethertap provides packet reception and transmission for user
space programs. It can be viewed as a simple Ethernet device,
which instead of receiving packets from a network wire, it
receives them from user space.
```

This is exactly what is needed to simulate an Ethernet subnet. On one side there is the Linux Ethertap device and on the other side the MipsSimulator Ethertap device. The communication between the two is made with the device file `/dev/tap0` (see figure C.1).

C.2 MipsSimulator Ethertap device (Ethertap.java)

The Ethertap device implemented in the MipsSimulator is a very simple memory mapped device. The memory layout of the device is described in table C.1.

The procedure to send data to the network (to Linux) is as follows:

- Proceed only if 'Send Data' (SD) is 0.
- Set the size of the packet to be sent in 'Data Out Length'.
- Fill the data in the 'Data Out' memory region.
- Set SD to 1.

The procedure to receive data from the network (from Linux) is as follows:

- When a new packet is received, interrupt 2 is triggered and 'Received Data' is set to 1.

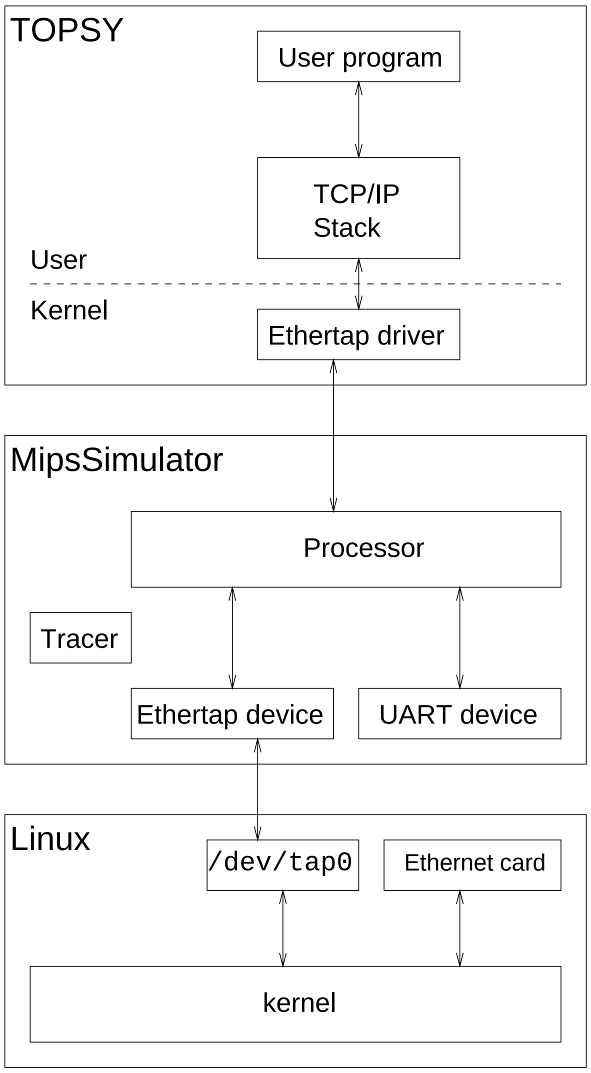

Figure C.1: Simulation environment

Offset	Description
0 (0x000)	Status register (1 byte, bit 0: Send Data, bit 1: Data Read)
4 (0x001)	Control register (1 byte, bit 0: Received Data)
16 (0x010)	Hardware Address (read only), 6 bytes
32 (0x020)	Data In Length (short)
36 (0x024)	Data In (device $\rightarrow$ Topsy)
2048 (0x800)	Data Out Length (short)
2052 (0x804)	Data Out (Topsy $\rightarrow$ device)

Table C.1: Ethertap device memory layout

- The size of the Ethernet packet can be read from ’Data In Length’.
- The packet can be read from the ’Data In’ memory region.
- Set ’Data Read’ to 1.

Unfortunately, java (JDK 1.1.6 and kaffe 0.9.0) seems to have problems reading and writing to special device files. Therefore an intermediate “Ethertap server” (ethertap.c) was written in C, which, as a side effect, makes it easier to port the simulation environment to other platforms.

This Ethertap server communicates the packets it receives from Linux via a TCP socket on the local machine on port 4000.

C.3 MipsSimulator Tracer (Tracer.java)

To simplify the evaluation of the performance and the debugging of the developed code in Topsy, a symbolic tracer was written which produces an output similar to this:

```
80023cbc                        schedulerSetBlocked {
80024328                          lock {
800215f8                            testAndSet (12/12)
                                  } lock (13/25)
80024350                          listSwap {
800208c4                            removeElem (19/19)
                                  } listSwap (39/58)
80024360                          unlock (2/2)
                                } schedulerSetBlocked (37/122)
```

The symbols indicate the name of the functions which the tracer recognises to be executed. The indenting and the curly braces are used to show the recursion level. The two numbers in parentheses indicate the number of cycles respectively without and with the called functions.

In this example, the ’lock’ function is called by ’schedulerSetBlocked’. The ’lock’ function calls ’testAndSet’, which is 12 cycles long. In total, the ’lock’ function is 25 cycles long (including the ’testAndSet’).

Note that there may be some cosmetic problems when a thread switch is executed, because the PC is changed by the scheduler. A ’>’ sign should appear to indicate such a jump.

C.4 MipsSimulator Profiler

A Profiler for the executed instructions was also written, so that performance analysis as made in appendix D were possible. The Profiler does function as follows:

- The MipsSimulator writes to a file the address of each executed instruction.
- A C program reads this file, reads the symbols and makes a statistic on how much instructions were executed for each function and how much times each function was called.

Appendix D

Measurements

D.1 Ping measurements (ICMP ECHO REQUEST and REPLY)

“calls” is the number of executed function calls, “ins” the number of executed instruction in that function and “i/c” is the average instructions count for that function.

	1000-bytes ping				**56-bytes ping**			
function	**calls**	**i/c**	**ins**	**%**	**calls**	**i/c**	**ins**	**%**
longCopy	16	265	4242	16.76	16	58	938	4.49
netipChecksum	3	482	1446	5.71	3	125	377	1.81
schedule	19	71	1355	5.35	19	71	1355	6.49
hashListGet	43	28	1245	4.92	43	28	1245	5.97
netattrGet	35	34	1215	4.80	35	34	1215	5.82
saveContext	22	53	1166	4.61	22	53	1166	5.59
msgDispatcher	21	54	1138	4.50	21	54	1138	5.45
tmMsgRecv	11	100	1100	4.35	11	110	1100	5.27
restoreContext	22	47	1034	4.09	22	47	1034	4.95
netattrSet	27	27	729	2.88	27	27	729	3.49
testAndSet	60	12	720	2.84	60	12	720	3.46
kSend	11	58	640	2.53	11	58	640	3.07
netattrNew	2	318	636	2.51	2	318	636	3.05
listSwap	16	39	627	2.48	16	39	627	3.00
syscallExceptionHandler	21	26	546	2.16	21	26	546	2.62
getMessageFromQueue	11	48	529	2.09	11	48	529	2.53
listGetFirst	35	14	515	2.03	35	14	515	2.47
netbufAlloc	3	156	468	1.85	3	137	411	1.97
hashFunction	43	9	387	1.53	43	9	387	1.85
lockTry	38	8	304	1.20	38	8	304	1.46
removeElem	16	19	304	1.20	16	19	304	1.46
listMoveToEnd	11	26	296	1.17	11	26	296	1.42
schedulerSetBlocked	8	37	296	1.17	8	37	296	1.42
schedulerSetReady	8	36	288	1.14	8	36	288	1.38
lock	22	13	286	1.13	22	13	286	1.37
kRecv	11	23	262	1.04	11	23	262	1.26
netethUp	1	228	228	0.90	1	228	228	1.09
netbufFreeBuf	3	65	196	0.77	3	62	186	0.89
addMessageInQueue	3	60	180	0.71	3	60	180	0.86
netethDown	1	149	149	0.59	1	149	149	0.71
netdbgPrintf	25	5	125	0.50	24	5	120	0.57
unlock	60	2	120	0.48	60	2	120	0.58

	1000-bytes ping				56-bytes ping			
function	**calls**	**i/c**	**ins**	**%**	**calls**	**i/c**	**ins**	**%**
tmMsgSend	10	12	120	0.47	10	12	120	0.58
tmIdleMain	60	1	119	0.47	60	1	119	0.57
netmodSendUp	3	38	114	0.45	3	38	114	0.55
netcfgSendNextDown	3	33	99	0.39	3	33	99	0.47
netifGetAttr	9	11	99	0.39	9	11	99	0.47
netcfgSendNextUp	3	33	99	0.39	3	33	99	0.47
memCopy	2	42	84	0.33	2	42	84	0.40
netipFillHeader	1	82	82	0.32	1	82	82	0.39
netmodSendDown	3	27	81	0.32	3	27	81	0.39
neticmpEchoUp	1	79	79	0.31	1	79	79	0.38
netipForUs	1	79	79	0.31	1	79	79	0.38
netipDownSend	1	78	78	0.31	1	78	78	0.37
netcfgarpResolveETHIP	1	72	72	0.28	1	72	72	0.35
ioDeviceMain	0	0	67	0.26	0	0	67	0.32
ioCheckBufferAddress	2	33	66	0.26	2	33	66	0.32
netcfgarpLookupETHIP	1	65	65	0.26	1	65	65	0.31
netcfgTransformDown	3	18	56	0.22	3	18	56	0.27
netbufLen	5	10	50	0.20	5	10	50	0.24
netcfgNextUp	3	15	47	0.19	3	15	47	0.23
netipVerify	1	45	45	0.18	1	45	45	0.22
neticmpUp	1	45	45	0.18	1	45	45	0.22
netbufFree	3	15	45	0.18	3	15	45	0.22
netipDownSendFragment	1	45	45	0.18	1	45	45	0.22
nettapRead	1	44	44	0.17	1	44	44	0.21
tmSetReturnValue	21	2	42	0.17	21	2	42	0.20
netbufAddHead	2	21	42	0.17	2	21	42	0.20
netipUp	1	41	41	0.16	1	41	41	0.20
nettapMain	0	0	41	0.16	0	0	41	0.20
netethMain	0	0	40	0.16	0	0	40	0.19
intDispatcher	1	39	39	0.15	1	39	39	0.19
ipcResetPendingFlag	19	2	38	0.15	19	2	38	0.18
genericSyscall	2	19	38	0.15	2	19	38	0.18
netipMain	0	0	36	0.14	0	0	36	0.17
netcfgNextDown	3	11	33	0.13	3	11	33	0.16
ethertap_read	1	33	33	0.13	1	33	33	0.16
netipSetAttrs	1	33	33	0.13	1	33	33	0.16
nettapDown	1	29	29	0.11	1	29	29	0.14
ethertap_write	1	28	28	0.11	1	28	28	0.13
ioWrite	1	26	26	0.10	1	26	26	0.12
netioWrite	1	26	26	0.10	1	26	26	0.12
ioRead	1	26	26	0.10	1	26	26	0.12
ethertap_interruptHandler	1	25	25	0.10	1	25	25	0.12
mmAddressSpaceRange	2	12	24	0.09	2	12	24	0.12
netipDown	1	21	21	0.08	1	21	21	0.10
netcfgNextUpIP	1	20	20	0.08	1	20	20	0.10
tmResetClockInterrupt	1	20	20	0.08	1	20	20	0.10
netcfgNextUpEth	1	18	18	0.07	1	18	18	0.09
neticmpMain	0	0	17	0.07	0	0	17	0.08
hwExceptionHandler	1	14	14	0.06	1	14	14	0.07
netcfgTransformUp	3	2	6	0.02	3	2	6	0.03
netipRoute	1	2	2	0.01	1	2	2	0.01

Bibliography

[1] Jose Brustoloni and Peter Steenkiste. User-level protocol servers with kernel-level performance. In *IEEE INFOCOM '98*, San Francisco, April 1998.

[2] G. Fankhauser, C. Conrad, E. Zitzler, and B. Plattner. Topsy - a teachable operating system, 1997.

[3] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, and T. A. Proebsting. Scout: A communications-oriented operating system. In *Hot OS*, May 1995.

[4] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Reading, Massachusetts, 1994.

[5] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, Reading, Massachusetts, 1994.